

Сергей
Бобровский

ТЕХНОЛОГИИ DELPHI 2006

Новые возможности

- Язык Delphi и шаблоны проектирования
- Доступ к СУБД по технологии BDP.NET
- Быстрое создание приложений ASP.NET и веб-служб
- Технология моделирования ECO III

Сергей Бобровский

ТЕХНОЛОГИИ
DELPHI 2006
Новые возможности



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск

2006

ББК 32.973-018.2
УДК 681.3.06
Б72

Бобровский С. И.
Б72 Технологии Delphi 2006. Новые возможности. — СПб.: Питер, 2006. — 288 с.: ил.

ISBN 5-469-01438-X

В книге рассмотрены новые возможности системы программирования Delphi 2006 и представлены нововведения в языке, оболочке, редакторе, компиляторе и отладчике. Объяснены новые технологии работы с базами данных и создания приложений для Интернета. Особое внимание уделено средствам и технологиям повышения эффективности работы программистов: рефакторингу, шаблонам проектирования, унифицированному языку UML и технологии моделирования ECO.

Издание рассчитано на программистов всех уровней подготовки. Начинающие разработчики познакомятся с удобными средствами быстрого создания программ, программисты средней квалификации изучат современные подходы к разработке сложных систем и новые средства визуального проектирования, а профессионалам будут интересны новые возможности технологий моделирования ECO III, UML 2.0 и паттернов проектирования.

ББК 32.973-018.2
УДК 681.3.06

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5-469-01438-X

© ЗАО Издательский дом «Питер», 2006
© Издательство «Развитие», 2006

Содержание

Введение	11
Что дальше?	12
Управление жизненным циклом приложений: технология ALM.....	13
Архитектура, управляемая моделью: технология MDA	14
UML-моделирование прикладных программ: технология ECO	16
Введение в платформу .NET	17
Из чего состоит .NET	17
Оболочка .NET Framework	19
Среда поддержки выполнения CLR.....	19
Технология доступа к данным ADO.NET	22
Технология создания Web-приложений ASP.NET.....	22
Поддержка Web-служб	23
Безопасность	24
Глава 1. Новые возможности оболочки	25
1.1. Технические требования и установка Delphi 2006.....	25
1.2. Главное окно	27
1.3. Список проектов	30
1.4. Репозитарий объектов.....	31
1.5. Палитра инструментов.....	31
1.6. Дизайнер	33
1.7. Адаптивный режим работы Дизайнера (Live Designer).....	34
1.8. HTML-дизайнер.....	35
1.9. Инспектор объектов.....	36
1.10. Менеджер проектов	36
1.11. Настройки среды	37

1.12. Новые компоненты интерфейса	39
1.12.1. Панель-сетка (TGridPanel)	39
1.12.2. Плавающая панель (TFlowPanel)	39
1.13. Разное	39
Глава 2. Технологии эффективного редактирования	40
2.1. Редактор Delphi	40
2.2. Визуализация кода	42
2.3. Построение модели кода (Model View)	42
2.4. Подсказки в коде (Code Insight).....	44
2.5. Шаблоны кода (Code Templates)	46
2.6. Умные блоки (Smart Block)	48
2.7. Средства компактного свертывания кода (Code Folding).....	48
2.8. Организация закладок	50
2.9. Синхронное редактирование (Code Sync).....	50
2.10. Ведение списка отложенных дел (ToDo).....	52
2.11. Быстрое комментирование.....	53
2.12. Расширенные комментарии.....	53
2.13. Клавиатурные макросы	54
Глава 3. Технологии интеллектуального редактирования (рефакторинга)	55
3.1. Поиск модуля (Find Unit)	55
3.2. Поиск ссылок (Find References)	56
3.3. Поиск класса (Find Class)	56
3.4. Быстрая декларация переменной и поля (Declare Variable, Declare New Field)	57
3.5. Визуальное создание суперкласса (Extract Superclass)	57
3.6. Визуальное создание метода (Extract Method).....	58
3.7. Визуальное изменение списка параметров процедуры (Change Parameters)	59
3.8. Глобальное переименование идентификатора (Rename)	59
3.9. Перемещение поля внутрь класса (Introduce Field/Variable)	60
3.10. Перемещение описания между классами (Move)	61
3.11. Перемещение членов между классами (Pull Members Up/Down)	61
3.12. Удаление лишних переменных (Inline Variable).....	61

7.5.9. Шаблон Strategy (Стратегия).....	188
7.5.10. Шаблон Template Method (Шаблонный метод).....	190
7.5.11. Шаблон Visitor (Посетитель).....	191
7.6. Обслуживание шаблонов.....	193
Глава 8. Технология моделирования с помощью языка UML	195
8.1. Унифицированный язык моделирования.....	195
8.2. Составные части диаграмм.....	196
8.3. Создание UML-проекта.....	197
8.4. Технология моделирования UML 1.5.....	197
8.4.1. Диаграмма классов (Class Diagram).....	197
Добавление класса на диаграмму Delphi	198
Свойства/атрибуты класса	199
Методы.....	201
Отношения.....	202
Интерфейсы	207
Объекты	207
8.4.2. Диаграмма вариантов использования (Use case diagram).....	208
Актеры.....	208
Варианты использования	209
Отношения между актерами и прецедентами	210
8.4.3. Диаграмма последовательности (Sequence diagram).....	213
Объекты	214
Сообщения.....	215
8.4.4. Диаграмма кооперации (Collaboration Diagram).....	218
8.4.5. Диаграмма состояний (Statechart Diagram).....	221
Автоматы	221
Состояния	222
Подавтоматы	223
8.4.6. Диаграмма деятельности (Activity Diagram)	224
8.4.7. Диаграмма компонентов (Component Diagram)	226
8.4.8. Диаграмма развертывания (Deployment Diagram)	228
8.4.9. Комментарии	228
8.4.10. Экспорт диаграмм.....	229
8.4.11. Гиперсвязи (Hyperlinks).....	230

6.3.2. Структура Web-службы.....	130
6.3.3. Технология создания простой Web-службы	130
6.3.4. Технология создания клиента Web-службы	135
Пример: подготовка описания интерфейса Web-службы.....	135
Пример: организация доступа к Web-сервису из кода.....	138
Пример: программирование выдачи сложных значений	139
Пример: обращение к реальным Web-службам.....	141
Глава 7. Технология шаблонов (паттернов) проектирования	144
7.1. Что такое шаблоны проектирования	144
7.2. Группы шаблонов.....	145
7.3. Порождающие шаблоны.....	146
7.3.1. Шаблон Abstract Factory (Абстрактная фабрика)	146
7.3.2. Шаблон Builder (Строитель)	150
7.3.3. Шаблон Factory Method (Фабричный метод).....	153
7.3.4. Шаблон Prototype (Прототип).....	156
7.3.5. Шаблон Singleton (Одиночка)	159
7.4. Структурные шаблоны.....	161
7.4.1. Шаблон Adapter (Адаптер).....	161
7.4.2. Шаблон Bridge (Мост)	163
7.4.3. Шаблон Composite (Компоновщик)	165
7.4.4. Шаблон Decorator (Декоратор)	167
7.4.5. Шаблон Facade (Фасад)	170
7.4.6. Шаблон Flyweight (Приспособленец).....	171
7.4.7. Шаблон Proxy (Прокси).....	174
7.5. Шаблоны поведения	175
7.5.1. Шаблон Chain of Responsibility (Цепочка обязанностей)	175
7.5.2. Шаблон Command (Команда).....	177
7.5.3. Шаблон Interpreter (Интерпретатор)	179
7.5.4. Шаблон Iterator (Итератор).....	180
7.5.5. Шаблон Mediator (Посредник).....	182
7.5.6. Шаблон Memento (Хранитель).....	183
7.5.7. Шаблон Observer (Наблюдатель)	185
7.5.8. Шаблон State (Состояние)	186

5.1.4. Способы вызова хранимых процедур	98
5.1.5. Другие возможности	100
5.2. Пример работы с базами данных dbGo (ADO.NET)	100
5.3. Технологии создания многоуровневых приложений баз данных.....	103
5.3.1. Технология создания многоуровневых BDP-приложений баз данных.....	104
Пример: создание сервера приложений	105
Пример: создание клиентского приложения	106
5.3.2. Создание многоуровневых приложений для платформы .NET с помощью технологии DataSnap	108
Пример: создание сервера приложений	108
Пример: доступ к серверу из приложений .NET	110
Глава 6. Технологии создания Web-приложений.....	113
6.1. Технологии создания приложений ASP.NET	113
6.1.1. Новые возможности Delphi по поддержке технологии ASP.NET	114
6.1.2. Структура приложения ASP.NET	114
6.1.3. Пример создания простого приложения ASP.NET	115
6.1.4. Работа с редактором кода HTML	117
6.2. Технологии ASP.NET по работе с базами данных	118
6.2.1. Технология стандартного соединения приложения ASP.NET с базами данных	118
Пример: заготовка серверного приложения	118
Пример: связывание данных	120
6.2.2. Технология DB Web.....	124
Пример: связь с базами данных с помощью технологии DB Web.....	124
Замечание: DB Web и XML.....	125
Пример: сохранение данных в XML-файле	126
Пример: загрузка данных из XML-файла	127
6.2.3. Рекомендации по настройке и отладке приложений ASP.NET	127
Замечание: сервер Cassini	127
6.3. Технологии создания Web-служб (Web Services)	129
6.3.1. Принцип быстрой разработки Web-служб	129

3.13. Безопасное удаление свойств и методов (Safe Delete)	62
3.14. Быстрое перемещение строки в ресурсы (Extract Resource String)	62
3.15. Отмена операций рефакторинга.....	63
Глава 4. Технологии компиляции и отладки	64
4.1. Новое в языке	64
4.1.1. Использование расширенного набора символов	64
4.1.2. Поддержка пространства имен	65
4.1.3. Дополнительные спецификаторы видимости.....	67
4.1.4. Абстрактные и закрытые классы	68
4.1.5. Помощники классов	68
4.1.6. Поля класса	69
4.1.7. Внутренние типы и константы классов.....	70
4.2. Новые возможности вызова подпрограмм.....	71
4.2.1. Начальные значения параметров функций	72
4.2.2. Динамические многомерные массивы	72
4.2.3. Перебор элементов контейнера.....	73
4.2.4. Перегрузка операций	74
4.2.5. Перегрузка преобразований типов.....	77
4.2.6. Типы данных .NET.....	79
4.2.7. Динамически размещаемые массивы .NET	80
4.2.8. Атрибуты классов.....	80
4.2.9. Управляемый и неуправляемый код	82
4.3. Технологии отладки: новые возможности	84
4.3.1. Точки прерывания	84
4.3.2. Исключительные ситуации	84
4.3.3. Смешанный код	85
4.3.4. Загрузка символических таблиц.....	87
4.3.5. Разное	87
Глава 5. Технологии работы с базами данных	90
5.1. Технология работы с базами данных BDP.NET	90
5.1.1. Визуальный Проводник данных (Data Explorer)	90
5.1.2. Технология Borland Data Providers for .NET (BDP.NET)	91
5.1.3. Пример создания приложений BDP.NET	93

8.5. Технологии моделирования UML 2.0.....	231
8.5.1. Что нового в UML 2.0.....	231
8.5.2. Диаграммы деятельности.....	232
8.5.3. Организация последовательных процессов.....	233
Поток управления.....	233
Поток данных.....	234
8.5.3. Диаграммы классов.....	235
8.5.4. Диаграммы компонентов.....	236
8.5.5. Диаграммы развертывания.....	237
8.5.6. Диаграммы вариантов использования.....	238
8.5.7. Диаграммы внутренней структуры (Composite Structure Diagram).....	239
8.5.8. Диаграммы последовательностей (Sequence Diagram).....	240
8.5.9. Диаграммы коммуникации (Communication Diagram).....	242
8.5.10. Диаграммы состояний (State Machine Diagram).....	242

Глава 9. ECO III : технология создания программ с помощью языка моделирования UML

9.1. Что такое ECO.....	244
9.2. Модель ECO.....	245
9.3. Объектное пространство ECO (EcoSpace).....	245
9.4. Этапы создания приложения ECO.....	246
9.5. Пример создания простого приложения ECO.....	247
9.5.1. Моделируем звездные системы.....	247
9.5.2. Создаем первый абстрактный класс.....	248
9.5.3. Строим иерархию классов.....	251
9.5.4. Добавляем дескрипторы ECO.....	252
9.5.5. Проектируем пользовательский интерфейс.....	253
9.5.6. Настраиваем элементы интерфейса.....	254
9.5.7. Построитель выражений OCL.....	255
9.5.8. Расширяем возможности приложения ECO.....	256
9.5.9. Автоформы.....	258
9.6. Организация связей между объектами ECO.....	258
9.6.1. Добавляем ассоциативную связь.....	258
9.6.2. Отслеживаем объект ECO, выбранный в таблице.....	259
9.6.3. Применяем автоформы к связанным таблицам.....	260

9.6.4. Настраиваем визуализируемые колонки.....	261
9.7. Доступ к модели ЕСО на уровне исходных текстов.....	262
9.7.1. Программное создание объектов ЕСО	263
9.7.2. Программное удаление объектов ЕСО.....	264
9.7.3. Программное связывание объектов ЕСО.....	266
9.8. Технология связи модели ЕСО с базой данных.....	268
9.8.1. Принципы использования СУБД в технологии ЕСО	268
9.8.2. Связываем приложение ЕСО с СУБД	268
9.8.3. Используем компоненты объектно-реляционной раскладки.....	269
9.8.4. Генерируем схему базы данных	269
9.8.5. Расширяем интерфейс пользователя.....	273
9.8.6. Синхронизация модели и базы данных: визуальная настройка	273
9.8.7. Использование файлов для хранения пространства ЕСО	273
9.8.8. Множественные соединения приложения ЕСО с СУБД	274
9.8.9. Технология создания модели ЕСО на основе существующей базы данных	275

Введение

Юбилейная, десятая по счету, начиная с 1995 года, версия системы программирования Delphi 2006 продолжает развитие стратегической линии компании Borland, направленной на интеграцию всех продуктов фирмы в рамках единой концепции управления жизненным циклом программного обеспечения. Можно уверенно утверждать, что новая версия системы несет самые масштабные изменения в продуктах и технологиях за всю историю развития Delphi. При этом поклонники системы, конечно, помнят, что и предыдущий релиз Delphi 2005 (Delphi 9) радикально отличался от Delphi 8 и являл собой глубокую переработку предшественника¹. В частности, в дополнение к компилятору для платформы .NET был восстановлен преждевременно отвергнутый компилятор для платформы Win32, что потребовало синхронизации версий библиотеки визуальных компонентов VCL. Также существенно обновилась технология ECO II и появились средства UML-моделирования из среды Borland Together.

Основные отличия Delphi 2006 от предыдущей версии связаны как с введением новых оригинальных технологий разработки, так и с продолжающимся объединением с другими продуктами Borland, прежде всего со средой моделирования Borland Together. В первую очередь важнейшие расширения системы относятся к поддержке UML-моделирования при построении прикладных программ. Значительное развитие также получила технология ECO, нацеленная на создание приложений посредством моделей, а также на стыковку с другими средствами Borland, рассчитанными на охват всего жизненного цикла разработки программного обеспечения.

В дополнение ко всем стандартным возможностям Microsoft .NET, доступ к которым в Delphi возможен как через оригинальные компоненты, так и через версию собственной библиотеки VCL, адаптированной для .NET,

¹ О темпах развития новых технологий и потребности рынка в их быстрой прикладной адаптации свидетельствует также и тот факт, что версия Delphi 8, в свою очередь, тоже была качественным шагом от среды Delphi 7, работающей в классической модели Windows 32, к исключительной поддержке платформы .NET. Однако практика показала, что потребность разработчиков в средствах создания приложений Win32 остается значимой, что и повлекло за собой возвращение к классической платформе в версиях Delphi 2005 и Delphi 2006.

программисту предлагаются компоненты собственной разработки Borland. Они, в частности, охватывают функции доступа к базам данных (концепция поставщиков данных Borland Data Providers, расширяющая стандартные механизмы .NET по работе с базами данных), причем не только по отношению к СУБД Microsoft SQL Server и Oracle, как в версии .NET 1.2, но и к СУБД Borland Interbase, IBM DB2 и Sybase. В дополнение к этим компонентам в среду включены утилиты взаимодействия с СУБД, а также оригинальный проектировщик меню в стиле Windows Forms.

Теперь на начальном этапе формирование требований заказчика к будущему проекту осуществляется через встроенное клиентское приложение системы CaliberRM. Проектирование пользовательского интерфейса происходит в Дизайнере Delphi, а работа с исходным текстом – в специализированном редакторе, который поддерживает рефакторинг (гибкую модификацию кода), синхронное редактирование одного файла несколькими разработчиками, наборы шаблонов кода, всевозможные подсказки и ускорители, другие средства. В среду Delphi также включены отладчик ASP.NET и набор утилит для развертывания приложений ASP.NET на Web-серверах.

Известно, что управление групповыми проектами эффективно организуется на базе систем отслеживания изменений и конфигурационного управления. Одна из них, Borland StarTeam (клиентская часть), встроена в Delphi.

Значительно повышена продуктивность работы с оболочкой среды программирования. Этому способствует гибкий редактор кода, средства интеллектуальной модификации исходных текстов (рефакторинга), появление большого числа шаблонов, существенно упрощающих и ускоряющих набор операторов.

Что дальше?

Осенью 2005 года корпорация Microsoft выпустила новую, вторую версию платформы .NET, которая является стратегической линией развития всего семейства технологий Windows. Delphi 2006 функционирует с использованием версии .NET 1.1, а в 2006 году появится версия Delphi с предварительным названием Highlander, которая предложит средства разработки 64-разрядных приложений для платформы .NET 2.0 и среды .NET Compact Framework, ориентированной на использование во встраиваемых системах и мобильных телефонах. При этом создание программ будет проводиться на основе единого набора компонентов библиотеки VCL, что позволит переносить приложения между платформами практически без изменения исходного кода.

В 2007 году ожидается выход новой версии Windows Vista, для которой появится среда Delphi for Vista с версией VCL для графической подсистемы Vista Avalon и подсистемы Web-служб Indigo. К тому же периоду относятся планы создания версии Delphi для программирования 64-разрядных приложений на языках Delphi и C++.

В более далекой перспективе, без сомнения, продолжится линия Borland на стыковку всего семейства продуктов поддержания жизненного цикла программного обеспечения и на быстрое включение новых технологий в среду за счет возможностей стандартной библиотеки VCL, обеспечивающей обратную совместимость исходных текстов.

Управление жизненным циклом приложений: технология ALM

Жизненный цикл приложения состоит из нескольких этапов. Сначала формируются требования к системе: они обычно вырабатываются в общении с заказчиком потенциальными покупателями. Затем начинается этап проектирования, когда готовятся модели будущего продукта и формируются его структуры, например структура баз данных. Однако само кодирование (или более общий процесс — непосредственная разработка программы) относится к следующему этапу жизненного цикла. Далее, когда приложение запрограммировано, его требуется протестировать в различных условиях эксплуатации, устранить выявленные ошибки, улучшить характеристики. Это происходит на этапе тестирования и отладки. И наконец, завершается жизненный цикл этапом развертывания приложения. Эта задача сама по себе бывает достаточно трудоемка, особенно если приложение взаимодействует с внешними базами данных, эксплуатируется на Web-сервере или реализовано в многоуровневой архитектуре.

Кроме того, все эти этапы пронизывает еще один процесс — процесс управления работой сотрудников, участвующих в реализации каждого из этапов. Хотя специфика их деятельности сильно различается, тем не менее, многие элементы управления на протяжении жизненного цикла остаются неизменными.

Продукты Borland полностью охватывают все этапы описанного выше жизненного цикла программного обеспечения. Для этого корпорация разработала концепцию Application Lifecycle Management (ALM, управление жизненным циклом приложения) и придерживается ее в качестве стратегического направления при создании и совершенствовании продуктов. Обработку требований на базе централизованного репозитория, их фиксацию, сопровождение, отслеживание изменений и взаимосвязей (в частности, связей с пользовательским интерфейсом), регистрацию истории версий и ряд других возможностей обеспечивает линейка Caliber. К положительным особенностям Caliber следует отнести хранение в общем репозитории всех сопроводительных документов и их связь с базой требований, что упрощает взаимную синхронизацию.

За построение моделей и проектирование архитектуры системы отвечает продукт Borland Together, который, начиная с версии Delphi 2006, встроен в среду, что подтверждает линию корпорации на полную интеграцию сво-

их решений. В качестве средств разработки программистам наряду с Delphi доступны JBuilder (среда программирования на языке Java), а также встроенные в среду Delphi система программирования C++Builder (она дополнена системой анализа ошибок времени выполнения CodeGuard) и средства создания в рамках Delphi программ для платформы .NET на языках C# (C#Builder) и даже VB.NET.

Процесс сквозного управления и контроля за версиями и изменениями обеспечивается системой коллективной работы StarTeam. Ее клиентская часть теперь тесно интегрирована в Delphi, и ее контролю поддаются не только исходные файлы с кодом, но и проекты, а также группы проектов. Для этого в контекстном меню соответствующих пунктов Менеджера проектов имеется раздел StarTeam с соответствующими возможностями.

Состыкован со StarTeam и Менеджер историй Delphi, что позволяет использовать возможности среды версионного контроля для учета изменений как на локальном компьютере, так и в рабочей группе.

Тестированием и оптимизацией созданных приложений занимается семейство серверных решений OptimizeIt, хранением данных — СУБД InterBase, а за развертывание, масштабирование и надежную работу ответственны: сервер приложений Borland Enterprise Server; технология Janeva, упрощающая стыковку приложений .NET, J2EE и CORBA; мобильная Java-СУБД JDataStore.

Популярная система тестирования DUnit для языка Delphi (или NUnit для языка C#) интегрирована в оболочку Delphi. Она автоматизирует процесс формирования проекта на базе тестов, используя ряд подходов из методологии экстремального программирования (www.xprogramming.ru), что подразумевает гибкое, адаптивное создание программы на основе тестов, которые выдвигает заказчик.

Система локализации теперь может работать в отдельном внутреннем окне Delphi, а ее настройки включены в раздел Translation Tools Options (Параметры среды локализации) общего редактора настроек, вызываемого командой Tools ➤ Options (Сервис ➤ Параметры).

При этом все продукты Borland открыты, совместимы с продуктами сторонних разработчиков и даже прямых конкурентов (так, выпущена версия Borland Together для среды Microsoft Visual Studio .NET) и ориентированы на самые разные платформы: Windows, .NET, Linux и др.

Архитектура, управляемая моделью: технология MDA

Погружаясь вглубь того или иного этапа жизненного цикла приложения (формирование требований, проектирование, программирование, тестирование, развертывание), можно обнаружить не менее оригинальные концепции организации работы специалистов: не столь масштабные, неже-

ли ALM, однако не менее эффективные и успешные. Так, тесно связанные этапы проектирования и разработки рекомендуется осуществлять в рамках идеологии Model Driven Architecture (архитектура, управляемая моделью, MDA). В отличие от ALM, концепция MDA была разработана не компанией Borland, а независимой некоммерческой организацией Open Management Group (консорциум по выработке стандартов объектного управления OMG, www.omg.org). Этот консорциум объединяет сотни компаний-разработчиков программного и аппаратного обеспечения. Технология MDA ориентирована на создание независимых от платформы и операционной системы и легко масштабируемых приложений — из готовых компонентов, которые могут использоваться повторно и многократно. При этом сам процесс разработки должен выполняться, как явствует из названия, под управлением модели: набора визуальных схем, описывающих внутреннюю структуру системы и принципы ее функционирования, не привязанного к конкретному языку или конкретной среде программирования. Такие схемы строятся с помощью унифицированного языка моделирования Unified Modelling Language (UML), который также был разработан консорциумом OMG для задач объектно-ориентированного проектирования.

Модели, созданные технологией UML, не связанные с конкретной системой разработки и выполнения, называются в терминологии MDA *платформно-независимыми моделями PIM (Platform Independent Model)*. Но MDA отнюдь не ограничивается построением абстрактных моделей и позволяет связать моделирование с конкретными средами разработки. Для этого существует понятие *моделей, специфичных для платформы*, — *PSM (Platform Specific Model)*. Различные коммерческие продукты (например, Delphi) выполняют перенос PIM-модели в определенную среду с ориентацией на заданную платформу (например, Windows или Java). При этом значительный объем итоговых исходных текстов, реализующих структуру и логику поведения UML-модели, генерируется автоматически.



PIM-модели сами по себе часто используются как готовые элементы сопроводительной документации.

Проектирование прикладной системы во многих случаях сводится к формированию и постепенному уточнению иерархии классов, а также к визуальному созданию структуры базы данных и привязки процедур ее модификации к пользовательскому интерфейсу. В принципе, в рамках MDA выполняются попытки моделировать и мелкие детали системы, например логику отдельных методов классов, однако такая деятельность требует определенного опыта и навыков. Она менее распространена в силу специфики и сложности, а также ограничений и неоднозначностей языка UML. В Delphi 2006 предпринята, пожалуй, одна из первых в программной индустрии попыток выпустить хоть и коммерческое, но весьма недорогое средство визуального построения логики программ. Это сделано в рамках технологии ECO III.

Такие важные возможности моделирования, только-только становящиеся доступными в рамках массовых продуктов программирования, связаны с поддержкой как генерации кода на основе моделей, так и обратного процесса реинжиниринга: создания моделей на базе исходных текстов и готовых структур баз данных. Подобные продукты на рынке, конечно, существуют, однако отличаются, как правило, весьма высокими ценами по сравнению со средами Borland Delphi и Borland Together.

UML-моделирование прикладных программ: технология ЕСО

Пожалуй, одно из самых радикальных улучшений Delphi 2006 в сравнении с прежними версиями — это расширенная поддержка технологии ЕСО (Enterprise Core Objects, ключевые корпоративные объекты), представленная теперь уже в виде третьей версии ЕСО III.

Каждый из ЕСО-компонентов представляет собой своеобразную «обертку» различных положений концепции MDA, промежуточный слой между средствами визуального проектирования программных моделей и их конкретной реализацией на некотором языке программирования. В данной книге рассматривается язык Delphi, а продукты Borland поддерживают кроме всего прочего ЕСО-компоненты для языка С#.

Технология ЕСО стала актуальной в связи с тем, что практика перевода требований заказчика напрямую в программный код на некотором языке программирования почти всегда страдает неполноценностью. Заказчик мыслит одними понятиями, программист — совсем другими, поэтому оба не подозревают о множестве потенциально возможных проблем. Так, заказчику некоторые моменты автоматизации выбранных процессов могут казаться очевидными, и поэтому он о них не упоминает, считая, что программа будет делать некоторый набор действий «сама». А программист подходит к созданию системы с узких технических позиций и редко учитывает все множество взаимосвязей между множеством требований, поэтому в середине проекта нередко выясняется, что архитектура решения не может быть легко дополнена требованиями, которые заказчику кажутся простыми и естественными. В результате приходится переделывать почти весь проект.

Идея промежуточного этапа формализации требований заказчика в виде визуальных моделей существует и развивается уже много лет. А когда удаётся ее как следует реализовать (в частности, в виде надстройки ЕСО), воплотив возможность прямой связи моделей с программным кодом «в обе стороны» (из моделей можно быстро и автоматически получить исходный код, а из исходного кода — набор визуальных моделей), то разработчику становятся доступны не только средства интерактивного согласования проекта с заказчиком, но и новый, очень мощный метод разработки приложений, открывающий качественно новые возможности. Он предоставляет высокоуровневые,

визуальные, удобные для человека средства построения формальной внутренней структуры программы и целостной архитектуры сложной информационной системы. (В Delphi этот процесс происходит в пределах так называемого ECO-пространства.) Работая на уровне модели, программист (чаще — системный аналитик) отвлекается от мелких деталей реализации, перестает думать в терминах отдельных операторов языка программирования и способен сразу охватить и продумать большие функциональные блоки и связи между ними. Отсюда и название предложенного Borland подхода MDA — архитектура (приложения), управляемая моделью. С ее помощью удастся быстро создавать объемные приложения разной функциональной направленности.

Отметим, что в системе Delphi 2006 возможности технологии ECO расширены на все типы приложений ASP.NET, в том числе и на Web-службы. Проектировщик пространства ECO (ECO Space Designer) приобрел полезную возможность выполнять объектно-реляционную раскладку для ранее созданных баз данных, сформированных вне рамок ECO-модели и, возможно, поддерживающих не только реляционную, но и объектную схемы. Другими словами, с помощью специального Мастера можно на основе существующей базы данных построить ее полноценную ECO-модель.

Еще одна удобная возможность ECO-проекта — сохранение ECO-модели в DLL-библиотеке с последующим ее подключением к другим проектам. За счет нового источника данных ECODataSource стало возможным создавать ECO-приложения на основе технологии взаимодействия с базами данных DB Web. Наконец, компонент PersistenceMapperProvider поддерживает взаимосвязь сразу с несколькими пространствами ECO — с помощью ведения защищенного пула соединения с несколькими базами данных.

Одно из важнейших нововведений ECO III — работа с так называемыми *машинами состояний* (State Machine), или автоматными схемами, которые позволяют описывать не только статические составляющие модели (структуры объектов), но и логику работы приложения, причем в визуальном режиме, с помощью специальных диаграмм. Для этого, в частности, был дополнен язык объектных ограничений OCL, который обычно применяется вместе с унифицированным языком моделирования UML. Версия OCL в Delphi 2006 получила название ECO Action Language. А редактор OCL-выражений теперь доступен из разных окон ECO-проекта, а также из Дизайнера форм и из Проектировщика UML-диаграмм.

Введение в платформу .NET

Из чего состоит .NET

Абсолютное большинство нововведений Delphi относится, конечно, к расширениям для платформы .NET. Известно, что платформа Microsoft .NET сегодня является ключевым стратегическим направлением развития опе-

рациональных систем корпорации Microsoft. Она столь обширна и включает столь большое число технологических концепций, что даже грядущая версия новой системы Windows Vista, которая первоначально планировалась к реализации полностью на платформе .NET, была-таки оставлена на ядре Windows XP SP2.

На момент написания этой книги платформа .NET существует в промежуточном варианте 1.1. Реализация ряда входящих в нее технологий еще не завершена, поэтому разработчики из Microsoft пока не смогли воспользоваться ее высоким потенциалом и в будущей системе Windows Vista ограничились менее универсальными подходами.



Вторая версия платформы .NET 2.0 вышла в конце 2005 г., и вариант Delphi для .NET 2.0 обещан корпорацией Borland уже в первом полугодии 2006 года.

К важнейшим особенностям и решениям, входящим в .NET, отнесем следующие (важные прежде всего с точки зрения разработчика):

- .NET Framework — технология программирования, ориентированная на быстрое визуальное создание надежных приложений. Включает множество стандартных классов и готовых компонентов для повторного использования;
- Common Language Runtime (CLR) — общезыковая среда выполнения, единая оболочка поддержки выполнения .NET-программ;
- ADO.NET — технология работы с базами данных;
- ASP.NET — система серверных сценариев для быстрого создания Web-приложений, основанная на популярной технологии активных серверных страниц Active Server Pages;
- Web-службы — средство обеспечения взаимодействия приложений по сети, основанное на XML-протоколе SOAP;
- встроенные технологии обеспечения безопасности работы и устойчивости приложений.

Важная особенность платформы .NET заключается в том, что она фактически представляет собой надстройку над низкоуровневыми операционными системами типа действующих версий Windows XP/2000 и при этом не зависит ни от типа операционной системы, ни от модели процессора. В результате среда .NET может быть реализована, например, как надстройка над системами Unix/Linux, и все написанные для нее программы будут без модификации исходных текстов работоспособны и на других платформах — потребуется лишь их перекомпиляция, да и то не всегда. Для обращения к операционной системе, на которой платформа .NET выполняется, в последнюю встроен специальный прямой программный интерфейс Platform Invocation Interface, который, впрочем, делает .NET-приложение менее надежным в плане обеспечения целого ряда правил безопасности и поэтому рекомендован к применению только в исключительных ситуациях. Так, популярные в ранних верси-

ях Delphi функции динамического выделения памяти наподобие GetMem в .NET-версии будут некорректными.

Рассмотрим перечисленные особенности .NET более подробно.

Оболочка .NET Framework

В состав этой оболочки включено несколько тысяч стандартных классов. Соответственно, оболочка ориентирована на поддержку объектно-ориентированной разработки. При этом она позволяет программистам добавлять свои собственные классы как на прикладном, так и на системном уровнях. Обычно в прикладных проектах разработчики создают оригинальные классы в качестве наследников стандартных классов, предлагаемых .NET Framework.

Библиотеки классов .NET Framework охватывают множество областей, связанных с функционированием операционной системы. В частности, программистам открываются следующие возможности .NET:

- организация работы в сети;
- обращение к базам данных;
- обеспечение режима безопасного функционирования;
- взаимодействие с системой ввода/вывода;
- использование готовых элементов пользовательского интерфейса (очень важная возможность, на основе которой удастся создавать компактные прикладные приложения, благо все базовые функции графических элементов управления поддерживаются на системном уровне и нет нужды включать их определения в свою программу, как это приходится делать при использовании библиотеки визуальных компонентов Delphi VCL);
- реализация множества стандартных возможностей, необходимых разработчикам: удобные типы данных от строк до массивов, средства их гибкой обработки, богатые наборы стандартных функций;
- поддержка интерфейса программирования Win32 для совместимости с действующими версиями Windows.

В прежних версиях Delphi (до 7-й включительно) практически вся поддержка подобных возможностей была реализована в собственной библиотеке Delphi VCL. При этом приложению приходилось включать в себя весь код соответствующих библиотек, и размер программ вырастал до весьма внушительных размеров (хотя несколько сотен килобайтов для современных систем — не величина). С появлением платформы .NET потребность в интеграции кода полностью отпала.

Среда поддержки выполнения CLR

Среда CLR, являющаяся частью оболочки .NET Framework, обеспечивает непосредственную работу программ, выполненных в соответствии с соглашениями .NET. Эта среда связывает программы, которые могут быть напи-

саны на любых языках, с операционным окружением .NET, обеспечивая и обслуживая все системные вызовы и подключая при необходимости те или иные стандартные библиотеки¹. Для этого, очевидно, двоичный код программы должен отвечать определенным требованиям.

Эти требования задаются низкоуровневым языком *Microsoft Intermediate Language (MSIL)*, который представляет собой своеобразный *виртуальный ассемблер, независимый от конкретной аппаратной платформы*. Именно благодаря такой технологии платформа .NET и может быть перенесена на другие операционные системы — для этого достаточно написать модуль поддержки в них ассемблера MSIL (модуль его трансляции в команды процессора конкретной модели).

Двоичный код любой .NET-программы представляет собой набор команд MSIL, в котором вся работа происходит с *единой системой типов данных Common Type System (CTS)* и обеспечиваются единые требования к безопасности (в частности, универсальным способом происходит проверка выхода индекса за границы массивов). Типичный .NET-компилятор создает результирующий MSIL-код из исходных текстов, записанных на любых поддерживаемых им языках программирования (например, Delphi, C++, C#). Этот код не сохраняет каких-либо особенностей исходного языка, а отвечает только требованиям технологий .NET и CLR. При этом, конечно, конкретная система программирования сможет охватить одни возможности .NET, но не реализовать другие в силу ограничений синтаксиса и семантики определенного языка программирования. Это возможно, например, из-за различий в поддержке инкапсуляции и видимости элементов классов. Нередко такая компиляция выполняется «на лету» (концепция *Just In Time*), непосредственно в ходе выполнения программы, потому что далеко не весь MSIL-код приложения будет обязательно выполняться и имеется возможность транслировать лишь те логические ветви, по которым происходит фактическое управление.

Итак, хотя в качестве средств разработки могут использоваться решения самых разных компаний, результатом их работы всегда будет двоичный MSIL-код, организованный по единой схеме взаимодействия с операционным окружением. В средствах разработки приложений для платформы Win32 ситуация была иной. Каждая среда обычно предлагала свое окружение и свою поддержку режима исполнения созданных с ее помощью программ. Так, с приложением Delphi необходимо было, как уже говорилось, поставлять собственные библиотеки VCL (или включать их в исходный код приложения), а для программ, созданных средствами Microsoft Visual C++, требовались библиотеки классов MFC. Разнились способы контроля выхода индексов за границы массивов и преобразования типов и многое другое.

¹ Разработчикам .NET-приложений доступен большой набор стандартных классов, через которые организуется взаимодействие программы с CLR.

Такие прикладные приложения, конечно, были несовместимы друг с другом, и организовать устойчивую работу и взаимодействие между комплексами, подготовленными разными разработчиками с помощью разных систем, было крайне сложно. А вот MSIL-код поддерживает средства обеспечения совместной работы и обеспечивает интеграцию программ, компонентов, готовых технологий безопасности и средств борьбы с ошибками.



Если приложение выполнено в полном соответствии с требованиями CLR, то его код называется управляемым.

В любое .NET-приложение включается метаописание как самого приложения, так и данных, которые оно использует. Это позволяет среде CLR работать со всеми данными наиболее оптимальным способом, представляя их в единой внутренней схеме. Благодаря этому, в частности, в CLR реализована автоматическая служба уборки мусора, очищающая области оперативной памяти, освобожденные после завершения приложения. Это решает все проблемы с утечкой памяти (появлением блоков памяти, не возвращенных операционной системе). Правда, не всегда управляемый код является безопасным, ведь, несмотря на все соглашения, программа в некоторый момент времени может выполнить запрещенную операцию (наподобие обращения к области памяти по неверному указателю на базе кода C++). Это может происходить в силу, например, ошибки прикладного разработчика и слишком гибких возможностей конкретного языка программирования. Чтобы избежать подобных недоразумений, универсальный код можно дополнительно проверить на предмет соответствия политике .NET-безопасности (набору правил), настроив его, в частности, на допустимый диапазон выполняемых MSIL-инструкций. Как правило, требуемые модели безопасности работы приложения задаются в исходных текстах с помощью специальных атрибутов, которые затем транслируются в метаописание создаваемого приложения. Поэтому отметим, что код, выданный компилятором MSIL, сам по себе не является управляемым. Степень его управляемости определяется метаинформацией.

На этапе выполнения программа представляется в виде так называемой «сборки». В нее входит метаинформация о всех выполняемых модулях (манифест сборки) и один или несколько исполнимых файлов и файлов динамически подключаемых DLL-библиотек со своей метаинформацией каждый. Понятие сборки введено для того, чтобы избежать замусоривания системы вспомогательными библиотеками, исключить проблемы, связанные с неверным использованием библиотек одного назначения, но разных версий, а также чтобы упростить возможность глобального или дистанционного обращения к элементам других сборок.

В состав оболочки .NET Framework входит библиотека классов, называемая Windows Forms (содержит формы графического интерфейса пользователя Windows и всевозможные элементы управления), сильно упрощающая

создание настольных приложений. В Delphi для этих целей применялась собственная библиотека VCL, однако версии, поддерживающие платформу .NET, обращаются уже к ее готовым ресурсам, поэтому и новый дизайнер форм выполнен в соответствии с типовыми рекомендациями и принципами .NET (в основном по аналогии с Microsoft Visual Studio .NET).

Отметим, что Delphi расширяет возможности .NET Framework такими технологиями, как собственные поставщики данных Borland Data Providers для .NET, упрощающие связь с СУБД InterBase, Oracle, IBM DB2, Sybase и Microsoft SQL Server, и наборами утилит для обслуживания БД. Кроме того, известная по прежним версиям Delphi библиотека визуальных компонентов VCL адаптирована для .NET. Существует в Delphi, конечно, и прямая поддержка компонентов Windows Forms.

Технология доступа к данным ADO.NET

Новая версия технологии Microsoft ADO.NET ориентирована на поддержку реляционных систем управления базами данных. В нее добавлена очень полезная возможность работы с базами в асинхронном режиме. Она требуется, например, когда связь сервера с клиентской частью организована через Интернет, а канал связи ненадежен. Однако работу всех частей системы при этом необходимо продолжать, и согласование (синхронизация) локальных и серверных данных выполняется в ADO.NET автоматически при очередном возобновлении связи.

В технологии ADO.NET применяются следующие понятия.

- *Источник данных (Data Source)*; фактически, конкретная технология или система обработки баз данных), который способен исполнять некоторые команды (как правило, SQL-инструкции). Нередко это обычная база данных или XML-файл, доступные через драйверы и оболочку управления.
- *Поставщик данных (Data Provider)*, который связывает прикладное приложение с СУБД и поддерживает обмен данных между ними.
- *Набор данных (Data Set)*; условная реляционная база данных), хранящийся в прикладной программе и внутренне представленный в формате XML, что открывает потенциальную возможность иерархической организации информации и упрощенного взаимодействия с Интернет-приложениями.

Многие классы ADO.NET дополнены и расширены в Delphi собственными классами и технологиями, ориентированными на типовые для этой среды средства создания приложений, взаимодействующих с базами данных (BDP.NET, BDE.NET).

Технология создания Web-приложений ASP.NET

Технология активных серверных страниц Active Server Pages (ASP) предложена корпорацией Microsoft уже достаточно давно. В версии .NET она рас-

ширена рядом удобных дополнений, повышающих скорость разработки. ASP-технология представляет собой систему выполнения на Web-сервере сценариев, написанных на различных языках программирования. Помимо сценарных команд в страницах ASP.NET можно при определенных условиях использовать команды языков C#, VB.NET или даже C++. Но наиболее важное отличие ASP.NET от прежних версий ASP — появление понятия Web-формы.

Web-форма, своеобразный аналог формы Windows, позволяет разрабатывать с помощью визуальных дизайнеров пользовательские формы, которые клиент видит в браузере при подключении к Web-серверу. Механизм взаимодействия с Web-формами с точки зрения разработчика Delphi напоминает схему взаимодействия обычного Windows-приложения с пользователем, что существенно упрощает процесс создания распределенных систем. Системная реализация таких механизмов, конечно, кардинально различается: события, генерируемые формой в браузере, обрабатываются другой программой, расположенной на сервере.

Программа-сценарий ASP.NET не интерпретируется, как это нередко реализовано в различных сценарных языках программирования, а предварительно транслируется в MSIL-код и потом исполняется на сервере, что позволяет пользоваться всеми преимуществами технологии .NET. Ранее в ASP-сценариях применялись преимущественно интерпретируемые языки с ограниченными возможностями наподобие VBScript.

Поддержка Web-служб

Технология ASP.NET поддерживает взаимодействие с Web-службами. Web-службы предоставляют дистанционный доступ пользователям и приложениям к своим открытым программным интерфейсам по высокоуровневому протоколу SOAP.



SOAP (Simple Object Access Protocol) — протокол обмена структурированными сообщениями в формате XML в распределенной вычислительной среде. Ранее технология SOAP применялась для дистанционного вызова процедур и методов объектов, функционирующих на других компьютерах, а сегодня активно задействована прежде всего в Web-службах и опирается на более низкий протокол HTTP для транспортировки сообщений.

При взаимодействии с Web-службами не важно, на каких платформах они функционируют и когда запущены. Можно стандартным способом, например, из любого браузера, получить XML-описание предоставляемого ими программного интерфейса в формате WSDL и затем дистанционно обратиться к любым его функциям и получить результат. В виде Web-служб реализованы, в частности, средства автоматического перевода Web-страниц на разные языки в поисковой системе Google.



WSDL (Web Services Description Language) — язык описания структуры Web-службы. Представляет собой XML-страницу, в которой содержится описание открытого интерфейса Web-службы, поясняющее, как вызывать тот или иной метод, какие нужны параметры, какие типы данных используются.

На одном сервере может существовать множество Web-служб. Так, на любом сайте может быть развернут свой набор таких дистанционных сервисов. Чтобы в них ориентироваться, была придумана технология UDDI.



UDDI (Universal Description, Discovery and Integration) — это стандарт, позволяющий организовывать в Интернете онлайн-каталоги услуг и упрощающий взаимный поиск клиентов и потребителей различных сервисов. Поиск выполняется с помощью SOAP-сообщений и основан на предоставлении доступа к WSDL-документам, описывающим соответствующие услуги.

На основе технологии UDDI разворачиваются так называемые *UDDI-регистры* — своеобразные тематические каталоги Web-служб. В частности, существуют *бизнес-регистры* UDDI, в которые можно занести собственную компанию и предоставляемые ею Web-услуги.

Технология ASP.NET предоставляет разработчику стандартные классы, описывающие различные элементы Web-служб. Программисту достаточно задать реализацию функций интерфейса его собственной, прикладной Web-службы, а остальное ASP.NET сделает автоматически. Отметим, что версия SOAP для платформы .NET отличается от версии SOAP, применяемой в Web-службах, — последняя представляет собой подмножество SOAP для платформы .NET, которая расширена, как это часто делает корпорация Microsoft, дополнительными средствами поддержки программного интерфейса Windows.

Безопасность

Концепция защиты доступа к коду *Code Access Security (CAS)*, пронизывающая все составные элементы технологии .NET, позволяет указывать в сборках допустимые к выполнению операции в коде, гибко настраивая его тем самым под требования общей политики корпоративной безопасности. При попытке нарушения прикладным кодом указанных правил (например, при попытке обращения к определенным файлам) вызывается механизм разрешения конфликтов, в котором, в частности, применяется привязка к полномочиям ролей пользователей. Роли распределяются в пределах платформы .NET и могут не совпадать с ролевым доступом на системном уровне Windows.

Глава 1. Новые возможности оболочки

В одной оболочке Delphi 2006 объединено сразу несколько компиляторов. Разработчик может создавать приложения не только на языке Delphi для платформ Win32 и .NET, но и на языке C# (для .NET), на языке C++ (для Win32), а также на языке Visual Basic .NET. Это определяет некоторые особенности процесса установки системы.

Значительно ускорена работа самой оболочки, визуальной среды разработки Delphi. Особенно это заметно при многократных загрузках среды в рамках одного сеанса Windows, а также в приложениях, где реализуется активная работа с оперативной памятью. В Delphi 2006 включен специальный менеджер памяти FastMM4, который отслеживает способы использования памяти и ее возможные «утечки».

1.1. Технические требования и установка Delphi 2006

Система Delphi 2006 работоспособна в операционных системах Microsoft Windows 2000, Microsoft Windows Server 2003 или Microsoft Windows XP Professional (рекомендуется). Компьютер должен быть оборудован ОЗУ от 256 Мбайт (рекомендуется 512 Мбайт) и процессором класса Pentium II 450 МГц (рекомендуется Pentium III 850 МГц). На жестком диске необходимо наличие 1,2 Гбайт свободного пространства. Перед началом установки программа-инсталлятор `install.exe` проверит, имеются ли на компьютере нужные системные и прикладные приложения для функционирования Delphi. Так, в текущей версии Windows должны быть установлены браузер Microsoft Internet Explorer v6.0 SP1, версия среды Microsoft .NET 1.1 и ее обновление .NET 1.1 SP1, библиотека разработчика .NET Framework SDK 1.1, XML-служба Microsoft XML Core Services (MSXML) 4.0 SP2 и среда Microsoft Visual J# .NET v1.1.

Если какой-либо из этих компонентов отсутствует, будет предложено их установить (все они имеются в дистрибутиве Delphi). Для этого дополнительно потребуется около 1 Гбайт пространства на системном разделе жесткого диска.



Рекомендуется также установить все последние обновления для Windows.

После того как операционная система будет приведена в соответствие с вышеуказанными требованиями к наличию программного обеспечения, заработает Мастер установки.

На первом этапе работы Мастера выбирается перечень устанавливаемых сред разработки (рис. 1.1).

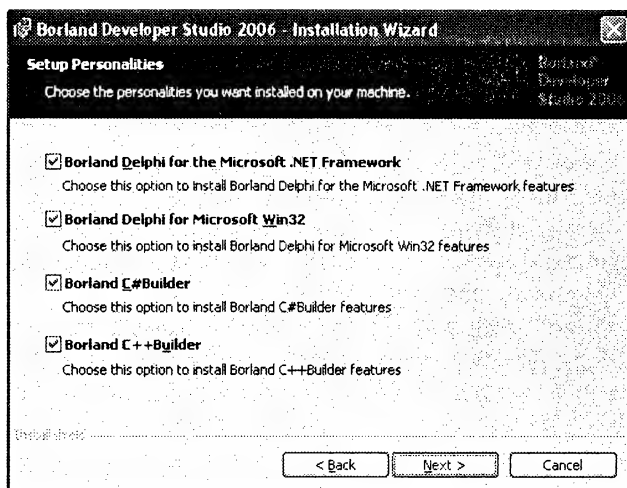


Рис. 1.1. Выбор устанавливаемых сред программирования

Допускается отдельная, выборочная установка компилятора Delphi для платформы .NET, компилятора Delphi для стандартных 32-разрядных Windows-приложений (Win32), компилятора языка C# для платформы .NET и компилятора языка C++ для платформы Win32. При этом они смогут вызываться как изнутри одной среды, так и порознь, через соответствующий пункт раздела Главного меню, который появится после установки.

Далее устанавливаются дополнительные компоненты пакета Borland Development Studio 2006 (рис. 1.2).

Установка флажка Install CaliberRM обеспечивает установку клиентской части среды CaliberRM по управлению требованиями к создаваемому программному продукту. Эта среда необходима в групповых проектах по созданию заказных систем.

Флажки Install RaveReports и Install ComponentOne Studio обеспечивают установку на компьютер соответственно набора компонентов RaveReports для создания отчетов и среды визуального проектирования таких отчетов, которые затем можно встраивать в свои приложения.

На следующем этапе определяется соответствие всевозможных расширений файлов приложениям комплекта Delphi. В некоторых случаях разра-

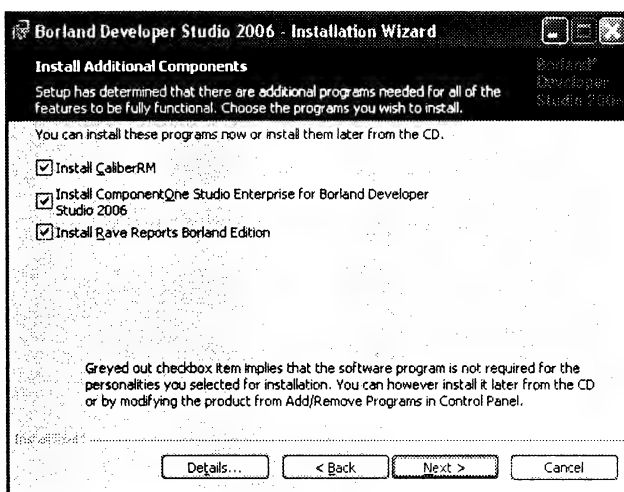


Рис. 1.2. Установка дополнительных компонентов пакета Borland Development Studio 2006

ботчики отказываются от связи Delphi с файлами программ на языках C++ (расширения .cpp, .h), C# (.cs) или со сценариями ASP.NET (.aspx), если уже используется другая среда разработки, например Microsoft Visual C++ или какие-либо редакторы Web-страниц.

После этого запускается процесс установки, который на современных компьютерах проходит весьма быстро.

1.2. Главное окно

После запуска среды пользователю предоставляется главное окно системы (рис. 1.3). Его характерная особенность заключается в центральной панели, которая основана на браузере Internet Explorer. Здесь сосредоточены ссылки на полезные ресурсы Интернета и локальные справочные документы по всем продуктам Borland. Здесь же, в разделе Recent Projects, будут отображаться ссылки на последние открывавшиеся проекты.

Дизайнер форм и редактор кода пока скрыты, так как никакого проекта мы еще не создали. Список доступных проектов можно получить командой File > New > Other (Файл > Создать > Другое), при этом вызывается так называемый Репозиторий объектов (см. ниже).

В правом нижнем углу главного окна расположена активно используемая палитра инструментов (Tool Palette), содержимое которой меняется в зависимости от текущего проекта. А исходно она представляет те же проекты, что строка меню. Вдобавок все они сгруппированы по темам (рис. 1.4).

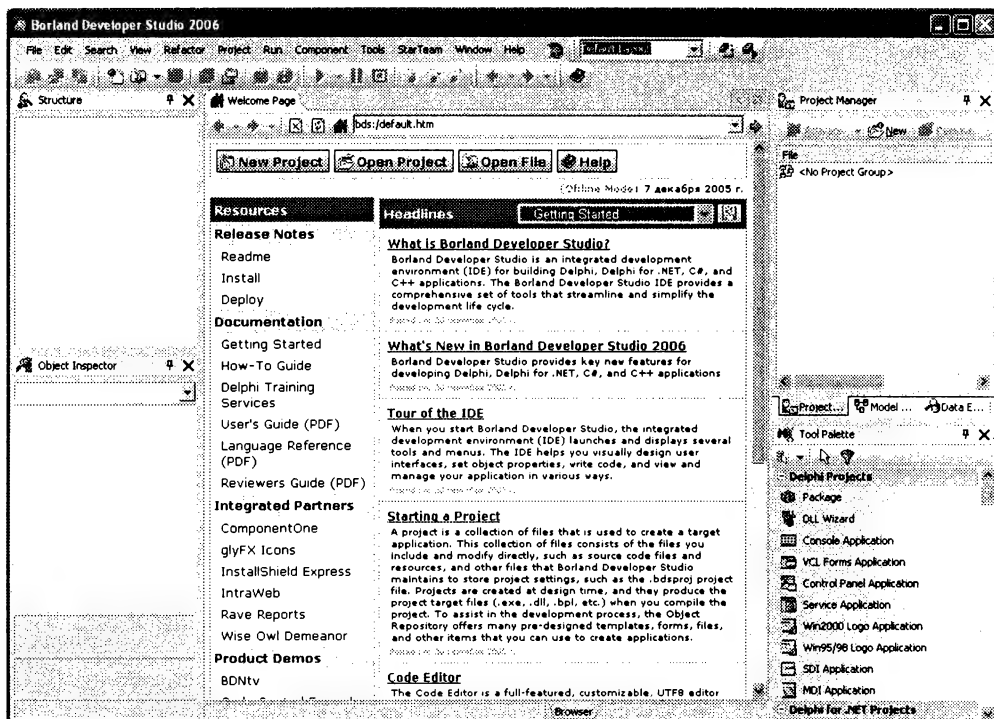


Рис. 1.3. Центральная панель основана на браузере Internet Explorer

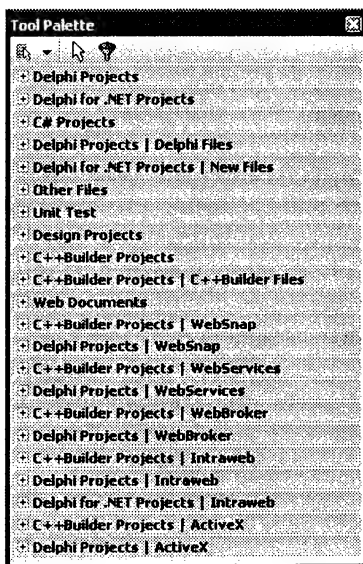


Рис. 1.4. Заготовки будущих проектов

Таблица 1.1. Разделы проектов системы Borland Developer Studio 2006

Раздел	Описание
Delphi Projects	VCL-приложения для платформы Win32
Delphi for .NET Projects	Основные виды приложений для платформы .NET, создаваемые на языке Delphi
C# Projects	.NET-приложения, разрабатываемые на языке C#
Delphi Projects/Delphi Files	Вспомогательные элементы проектов Delphi для платформы Win32
Delphi for .NET Projects/New Files	Вспомогательные элементы проектов Delphi для платформы .NET
Other Files	Различные элементы проектов Delphi и приложения на языке Visual Basic.NET
Unit Test	Организация тестовых проектов для ускоренного прототипирования программ
Design Projects	Построение UML-моделей
C++Builder Projects	Приложения, разрабатываемые на языке C++ для платформы Win32
C++Builder Projects/C++Builder Files	Вспомогательные элементы для приложений C++/Win32
Web Documents	Web-документы во всевозможных форматах (XML, HTML, JavaScript и другие)
C++Builder Projects/WebSnap	Создание WebSnap-приложений на языке C++
Delphi Projects/WebSnap	Создание WebSnap-приложений на языке Delphi
C++Builder Projects/WebServices	Создание Web-служб на языке C++
Delphi Projects/WebServices	Создание Web-служб на языке Delphi
C++Builder Projects/WebBroker	Создание кросс-платформных Интернет-приложений на языке C++
Delphi Projects/WebBroker	Создание кросс-платформных Интернет-приложений на языке Delphi
C++Builder Projects/Intraweb	Реализация технологии IntraWeb для платформы .NET на языке C++
Delphi Projects/Intraweb	Реализация технологии IntraWeb для платформы Win32 на языке Delphi
Delphi for .NET Projects/Intraweb	Реализация технологии IntraWeb для платформы .NET на языке Delphi
Delphi Projects/ActiveX	Компоненты ActiveX, создаваемые с помощью языка Delphi
C++Builder Files/ActiveX	Компоненты ActiveX на языке C++

1.3. Список проектов

Как видно, одним из заметных нововведений Delphi 2006 стало включение в среду системы программирования на языке C/C++, известной как Borland C++Builder. Она позволяет программистам, работающим на языке C/C++, готовить приложения Win32 с использованием библиотеки VCL и новых комфортных возможностей текущей среды.

Разработчикам приложений для платформы .NET доступна, конечно, возможность программирования на языке Delphi (как с использованием набора компонентов VCL.NET, так и без него, на основе стандартных классов Windows Forms и .NET Framework), а также на языке C#. Кроме того, можно создавать приложения и на языке Visual Basic .NET, но только кодировать всю программу от начала до конца придется вручную — Дизайнер форм для VB.NET в Delphi не предусмотрен.

В раздел проектов .NET включены две технологии, принципиально важные с точки зрения долгосрочной перспективы:

- ASP.NET, на базе которой в Delphi можно разрабатывать как приложения Web-форм, так и Web-сервисы;
- ECO III, которая включила расширенные средства связи UML-модели приложения с программным кодом и средства автоматического сохранения состояния модели в различных базах данных. Кроме того, в ECO III появилась принципиально новая возможность формирования с помощью UML не только статических аспектов модели, но и логики ее работы. Для этого используется машина состояний. Доступные в Delphi средства построения моделей программного обеспечения пополнились также встроенной системой моделирования Borland Together.

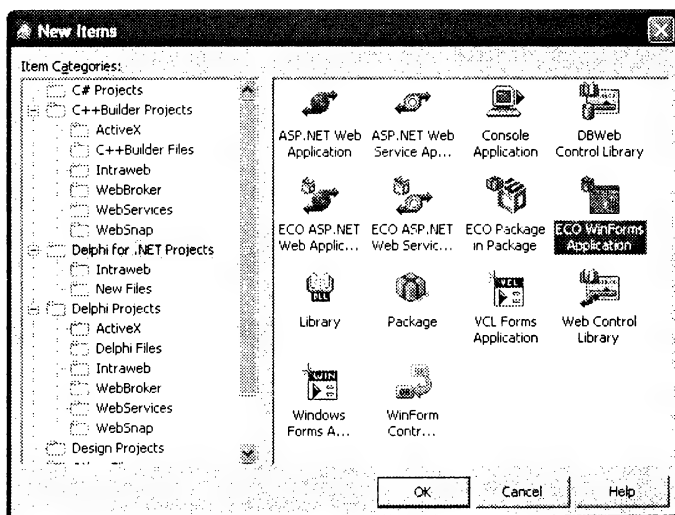


Рис. 1.5. Репозиторий объектов Delphi

1.4. Репозиторий объектов

В среде Delphi имеется удобная и важная возможность ускорения процесса разработки за счет размещения всевозможных шаблонов и заготовок проектов и форм в едином хранилище, называемом *Репозитарием объектов* (*Object Repository*). Доступ к его содержимому выполняется командой File ➤ New ➤ Other (Файл ➤ Создать ➤ Другое) — рис. 1.5.

Любой из элементов Репозитария можно модифицировать или удалить через контекстное меню, ввести новые категории проектов, а при желании содержимое можно легко расширить, добавляя текущий проект в качестве одного из готовых шаблонов с оригинальным именем. Для этого, доведя текущий проект до некоторого желаемого состояния, надо дать команду Project ➤ Add to Repository (Проект ➤ Добавить в Репозиторий).

1.5. Палитра инструментов

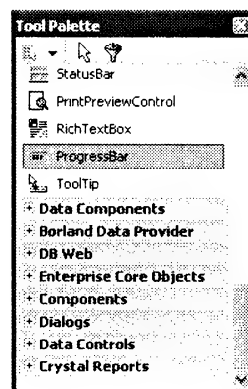
Создать любой проект (например, приложение Windows Forms) можно, дважды щелкнув мышью на строке с соответствующим названием в палитре инструментов. При этом на экране появится окно Дизайнера (активна закладка Design в нижней части центрального окна), а переключиться в редактор можно через закладку Code.

После открытия Дизайнера автоматически меняются и содержимое палитры инструментов: на ней появляются группы компонентов, доступные для размещения на форме и в окне Дизайнера в рамках текущего проекта.

Среди удобных нововведений в палитре инструментов отметим возможность автоматического размещения выбранного компонента на форме двойным щелчком на его названии в палитре. При этом, конечно, не стоит рассчитывать, что автоматическое размещение будет эстетичным. Зато исключается необходимость в дополнительном щелчке в Дизайнере. Это особенно удобно при выборе невизуальных компонентов.

В разделе Tool Palette (Tools ➤ Options) можно настраивать различные элементы оформления панели: размеры значков компонентов (Button Size), режимы размещения групп компонентов и их автоматического сворачивания (Auto Collapse Categories), режим вертикального расположения заголовков (Vertical Category Captions), а также цветовые схемы (подраздел Colors).

Внутри палитра инструментов разделена на тематические группы элементов (категории). Для быстрого поиска нужного элемента достаточно выде-



лить палитру мышью и начать ввод названия искомого компонента (например «eco... »).

Список внутри палитры будет прокручиваться к подходящей строке автоматически, а перечень доступных в ней элементов будет сокращаться до тех, которые соответствуют набранному тексту (он показывается в заголовке палитры). Отменить режим автоввода можно клавишей Esc.

В палитре инструментов появилась кнопка фильтрации (рис. 1.6).

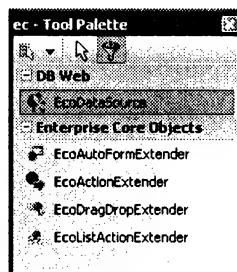


Рис. 1.6. Кнопка фильтрации компонентов

Если кнопка фильтрации нажата, то при наборе текста происходит автоматический отбор тех компонентов (элементов списка), которые начинаются с введенных символов (рис. 1.7).

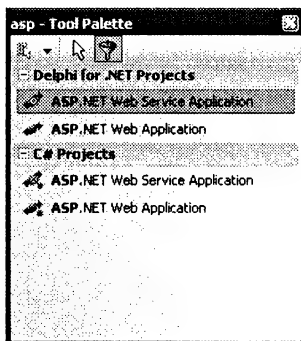


Рис. 1.7. Автоматический отбор компонентов

Набранный текст отображается в строке заголовка окна перед словами «Tool Palette». Для возврата к обычному режиму надо повторно нажать на кнопку фильтрации.

Порядок элементов можно настраивать самостоятельно. Для этого достаточно перетащить любой элемент в новое место списка. Аналогично меняется и порядок списка категорий.

Допускается добавление в палитру собственных категорий и компонентов, а также включение в нее шаблонов – целых групп компонентов, расположенных на форме некоторым удобным образом. Для этого надо в контекстном меню палитры выбрать пункт **Add New Category** (Добавить новую категорию) и в диалоговом окне ввести название новой категории. Элементы в нее можно добавлять перетаскиванием из других категорий палитры. Кроме того, выделив часть элементов на форме, их можно объединить в одном шаблоне командой **Component > Create Component Template** (Компонент > Создать шаблон компонента), после чего в диалоговом окне указать название нового компонента, выбрать категорию палитры инструментов и значок для нового компонента. В дальнейшем к нему можно будет обращаться так же, как и к любым другим элементам палитры.

1.6. Дизайнер

Приложения Windows создаются в Delphi 2006 аналогично тому, как это принято в старых версиях. Однако существует ряд отличий, связанных с существенным изменением и переработкой самой среды визуального программирования. Работа с формами – один из ключевых элементов подобных сред. Разработчик создает и настраивает внешний вид формы и всех связанных с ней элементов управления в Дизайнере – модуле визуального проектирования пользовательского интерфейса. При этом он использует шаблоны, хранящиеся в палитре инструментов.



Невизуальные компоненты проектов для платформы .NET (в отличие от проектов Win32) размещаются не в пределах проектируемой формы, как было в Delphi 7 и более ранних версиях, а в нижней части окна Дизайнера за пределами формы.

Элементы располагаются на форме в зависимости от настроек (**Tools > Options > Windows Forms Designer** (Сервис > Параметры > Дизайнер Windows Forms) – рис. 1.8).

Значения полей **Grid Size** определяют размер (шаг) опорной сетки – по ней выравниваются элементы управления. Флажок **Show grid** позволяет отключать отображение сетки, а флажок **Snap to grid** включает и выключает режим автоматического притягивания элементов формы к узлам сетки.

Практически идентичные режимы работы Дизайнера применяются при создании любых форм, поддерживаемых Delphi, – как форм Windows платформы .NET, так и Web-форм для технологии ASP.NET, а также стандартных форм Windows (VCL-форм) платформы Win32 и их .NET-версии VCL.NET.

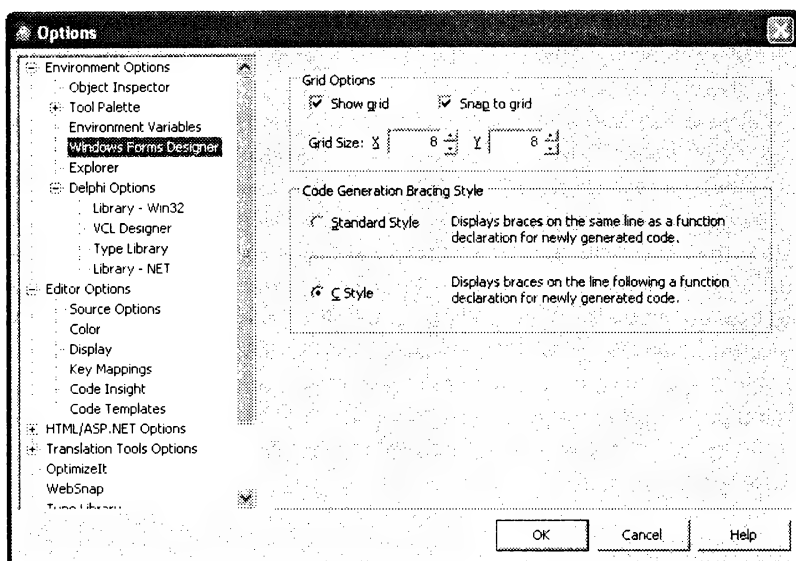


Рис. 1.8. Настройка параметров опорной сетки для Дизайнера форм

При создании VCL-формы встроенный в Delphi Дизайнер (когда он доступен через закладку Design) можно отключать. При этом Delphi начинает работать в режиме, привычном для пользователей Delphi 7, когда проектируемая форма показывается сама по себе, а окно редактора — само по себе, как бы вне рамок среды. Чтобы включить такой режим, надо снять флажок Embedder Designer в окне Tools ➤ Options ➤ VCL Designer (Сервис ➤ Параметры ➤ Дизайнер VCL).

1.7. Адаптивный режим работы Дизайнера (Live Designer)

При создании в Дизайнере форм проектов VCL или VCL.NET доступен режим так называемых направляющих линий *Designer Guidelines*. Он по умолчанию включен, а настраивается флажком Use designer guidelines (Использовать направляющие в Дизайнере) в свойствах среды: Tools ➤ Options ➤ Environment Options ➤ Delphi Options ➤ VCL Designer (Сервис ➤ Параметры ➤ Параметры среды ➤ Параметры Delphi ➤ Дизайнер VCL). В таком режиме перемещение компонентов на форме сопровождается появлением направляющих линий. Это упрощает выравнивание графических элементов, что нередко выполняется «на глазок». Работа по направляющим особенно удобна, если шаг сетки Дизайнера выбран небольшим, а компонентов на форме много.

Специальные метки отмечают средние точки перетаскиваемых объектов по отношению к близлежащим элементам. Они возникают, когда перемеща-

емый компонент приблизился к неподвижному объекту на расстояние, меньшее, чем задано в свойстве `Margins` (Поля). Чтобы выровнять элемент по его внешним границам, удобно использовать свойство `AlignWithMargins`.

Для компонентов-контейнеров (`TPanel`, `TFrame`) введено также свойство `Padding`, которое определяет, сколь близко к границам могут приближаться расположенные внутри них элементы.



В нижней правой части Дизайнера VCL-форм расположен небольшой значок (своеобразная глобальная карта), напоминающий, где и как на экране располагаются формы проекта.

1.8. HTML-дизайнер

Web-формы часто требуют особого дизайна, специфичного для HTML-страниц. Для его создания предназначен еще один режим работы Дизайнера: HTML-дизайнер. Он доступен, когда создается приложение ASP.NET, в которое помимо исходного текста на языке Delphi включается также исходный текст сценария на языке ASP.NET (или одном из других, входящих в категорию проектов Web Documents). Тогда соответствующий файл ASP.NET (с расширением `.aspx`) будет доступен на отдельной вкладке, а в редакторе его синтаксис будет подсвечен в соответствии с правилами соответствующего языка разметки.

Сам редактор кода HTML/ASP.NET расширен различными усовершенствованиями, повышающими скорость и комфортность обработки файлов стилей CSS, текстов HTML и тегов, а структура документа HTML может отображаться в иерархическом виде в Просмотрщике структуры.

В Delphi также встроена новая система форматирования документов HTML, базирующаяся на общедоступном решении Tidy (tidy.sourceforge.net). Оно разработано при поддержке консорциума WWW, вырабатывающего всевозможные стандарты для Сети, поэтому представление результирующего документа HTML будет максимально приближено к официально рекомендованному.

Чтобы включить режим форматирования средствами Tidy, надо перейти к настройкам Delphi: `Tools` ➤ `Options` ➤ `HTML/ASP.NET Options` ➤ `HTML Tidy Options` (`Сервис` ➤ `Параметры` ➤ `Параметры HTML/ASP.NET` ➤ `Параметры HTML Tidy`), где представлено множество средств настройки этого визуализатора, и установить флажок `Use HTML Tidy as the default formatter` (Использовать HTML Tidy как средство форматирования по умолчанию).

Проектировщику пользовательского Web-интерфейса теперь доступны два новых компонента: `HTML Flow Panel` и `HTML Grid Panel` из группы `HTML Elements` палитры инструментов. Они позволяют создавать на форме панели, элементы внутри которых располагаются соответственно либо в предопределенных позициях, либо в ячейках таблицы.

1.9. Инспектор объектов

В левой части главного окна Delphi обычно располагается окно Инспектора объектов (Object Inspector). Его содержимое меняется в зависимости от того, какой объект в Дизайнере выбран в данный момент.

Содержимое окна поделено на тематические группы свойств, а на закладке Events, как обычно, приводится список доступных событий для обработки. Некоторые компоненты (прежде всего невидимые) имеют встроенные редакторы своих свойств. Так, компоненты связи с базами данных обычно содержат внутренние редакторы формирования SQL-запросов. Теперь все редакторы доступны в нижней части Инспектора в виде «гиперссылок» (рис. 1.9).

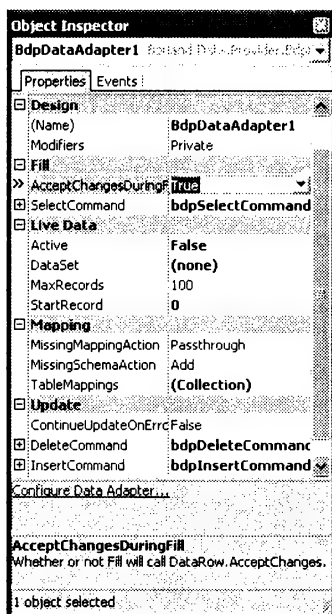
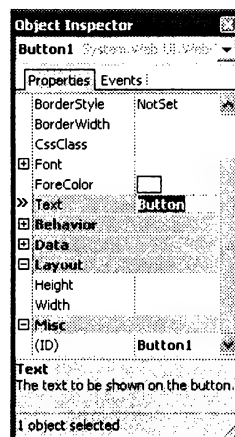


Рис. 1.9. Инспектор объектов в действии

1.10. Менеджер проектов

Этот традиционный для всех версий Delphi элемент среды предназначен для визуального сопровождения проекта. Менеджер представляет дерево файлов описаний форм и исходных текстов, исполнимого кода, библиотек. Несколько проектов можно объединить в группу, чтобы быстрее переключаться между

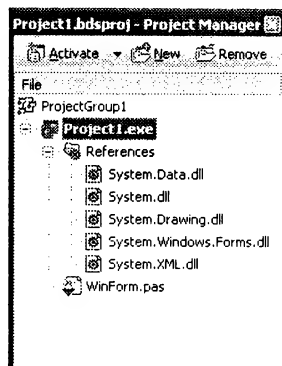
проектами. Находится окно Менеджера проектов по умолчанию в правой верхней части окна Delphi.

На этапе развертывания проекта каждая сборка .NET (фактически, файл .dll) должна размещаться либо в локальных каталогах проекта, либо в так называемом *глобальном кэше сборки* — *Global Assembly Cache (GAC)*. В нем одновременно хранятся разные версии одной сборки, а любая из сборок GAC считается общедоступной на данном компьютере.

Указать местонахождение сборки можно, выбрав в контекстном меню сборки в Менеджере пункт *Copy Local* (Копировать локально) или задав в Инспекторе объектов значение ее свойства *Copy Local*. Данный пункт меню просто отметится флажком. А если он не отмечен (значение свойства *Copy Local* равно *False*), значит, сборка будет храниться в глобальном кэше.

При необходимости в создаваемое сетевое клиентское приложение можно добавить Web-ссылку на некоторую Web-службу, используемую в нем. Для этого в контекстном меню дерева проектов надо выбрать пункт *Add Web Reference* (Добавить Web-ссылку), и в специальном браузере Borland UDDI Browser указать местонахождение WSDL-файла, описывающего нужный сервис, или просто задать адрес нужной службы. Описание соответствующего ей программного интерфейса на Паскале будет сгенерировано Delphi автоматически и добавлено в проект.

Менеджер проекта по умолчанию совмещен в одном окне с Просмотрщиком моделей (Model View), визуальным демонстрирующим структуру модели создаваемого приложения, и с Проводником данных (Data Explorer), предназначенным для визуального взаимодействия с СУБД. Проводник позволяет устанавливать действующие соединения с различными базами данных, тесно интегрирован с Дизайнером (любую из таблиц установленного соединения можно, например, просто перетащить мышью на форму, и соответствующие компоненты, обеспечивающие связь с базой, будут добавлены и настроены автоматически) и предлагает ряд дополнительных возможностей по настройке соединений.



1.11. Настройки среды

Среда Delphi выполнена в стиле, напоминающем классический интерфейс Microsoft Visual Studio, поэтому многие приемы работы с нею приведены в соответствие с неформальным стандартом Microsoft на средства разработки для платформы .NET (что, без сомнения, весьма удобно для программиста: не надо изучать новые интерфейсы). Так, все свойства Delphi доступны через команду главного меню *Tools > Options* (Сервис > Параметры), и если ранее они

представлялись в виде окна с множеством закладок, то в последних версиях Delphi окно настроек представляет собой панель категорий (в левой части окна), а справа — панель параметров, соответствующих выбранному элементу левой части (рис. 1.10).

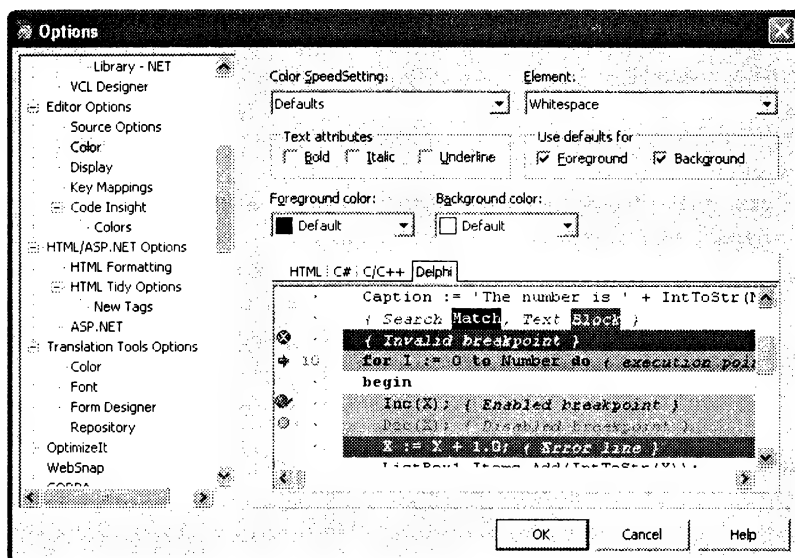


Рис. 1.10. Цветовые настройки

Таблица 1.2. Основные категории окна настроек

Категория	Описание
Environment Options	Параметры общих настроек среды
Editor Options	Параметры редактора исходных текстов
HTML/ASP.NET Options	Параметры редактора HTML и средств работы с технологией ASP.NET
OptimizeIt	Параметры системы оптимизации кода
WebSnap	Параметры технологии WebSnap
CORBA	Параметры технологии CORBA
ECO General Options	Параметры технологии ECO
Together	Параметры встроенной среды моделирования Together
Translate Tools Options	Параметры средств локализации
Debugger Options	Параметры встроенного отладчика

1.12. Новые компоненты интерфейса

1.12.1. Панель-сетка (TGridPanel)

Панель-сетка (группа Additional палитры инструментов) позволяет пропорционально размещать на форме элементы пользовательского интерфейса. По умолчанию панель-сетка имеет четыре раздела, обозначенных пунктиром. В каждый из них можно поместить только один элемент, а при увеличении числа элементов для них автоматически создаются новые разделы. При этом все разделы равны по размерам и по форме, благодаря чему они располагаются с равными интервалами, независимо от размера формы.

Так, разместив на панели-сетке несколько элементов, можно выровнять ее по размеру клиентского окна (главной формы), указав в свойстве `Align` значение `alClient`, и запустить приложение. Если теперь изменять размеры формы, то расположение элементов на ней будет синхронно модифицироваться с соблюдением пропорций.

1.12.2. Плавающая панель (TFlowPanel)

Плавающая панель (группа Additional палитры инструментов) обеспечивает автоматическое размещение элементов управления в своих рамках, начиная с левого верхнего угла. Каждый новый элемент добавляется вплотную к последнему установленному. Впоследствии их расположение будет меняться автоматически в зависимости от текущего размера плавающей панели.

1.13. Разное

Компоненты `TEdit`, `TComboBox`, `TBitBtn` и `TButton` из библиотек `VCL` и `VCL.NET` дополнились свойством `Align`, задающим способ выравнивания объекта относительно родительского окна. А компонент `TTrayIcon` позволяет задавать анимированный значок приложения, показываемый на панели инструментов, когда приложение свернуто пользователем. Время анимации в миллисекундах задается в свойстве `AnimateInterval`, сам значок — в свойстве `Icon`.

Реализована в Delphi 2006 поддержка технологии «*pan and scroll*» для новых мышей Microsoft Intellimouse, допускающих прокручивание не только по вертикали, но и по горизонтали.

Глава 2. Технологии эффективного редактирования

2.1. Редактор Delphi

Редактор Delphi содержит множество интересных возможностей, которые даже в таком простом на первый взгляд процессе, как подготовка исходного текста, способны существенно повысить производительность работы: в разы, а может быть, даже на порядок! Так, в него добавлены новые средства рефакторинга, подробно описанные в соответствующем разделе книги. Например, режим *Sync Edit* позволяет одновременно править все экземпляры одного идентификатора в коде, а одной командой можно прокомментировать большой блок исходного текста.

Работа с исходным текстом в Delphi уже давно не ограничена одним редактором. Так, Просмотрщик структуры программы (Structure View) представляет в виде дерева иерархическую структуру кода: классы с полями и методами, что позволяет двойным щелчком быстро переходить к их определениям и реализациям в исходном коде.

Отметим также популярную функцию поиска описаний различных понятий в исходном тексте и навигации по ним: при нажатой клавише CTRL курсор следует подвести к выбранному определению, после чего появляется всплывающая подсказка. Щелчок на ней открывает файл с текстом соответствующего описания, или же текущий файл прокручивается к точке декларации данного идентификатора.

При вводе программного кода возможен анализ его синтаксической корректности в реальном времени. Он выполняется, если включен флажок *Enable Error Insight* в настройках редактора (Tools > Options > Editor Options > Code Insight, Сервис > Параметры > Параметры редактора > Подсказки кода). При обнаружении ошибки соответствующее место в редакторе подчеркивается волнистой красной линией, а при наведении на него указателя мыши возникает всплывающая подсказка с описанием причины проблемы.

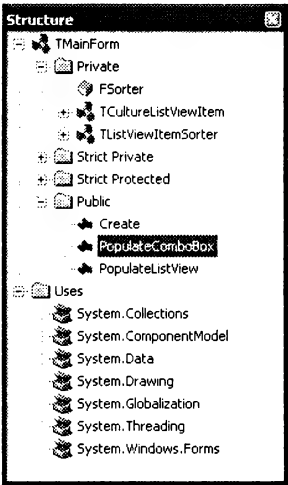


Рис. 2.1. Средство просмотра структуры исходного текста

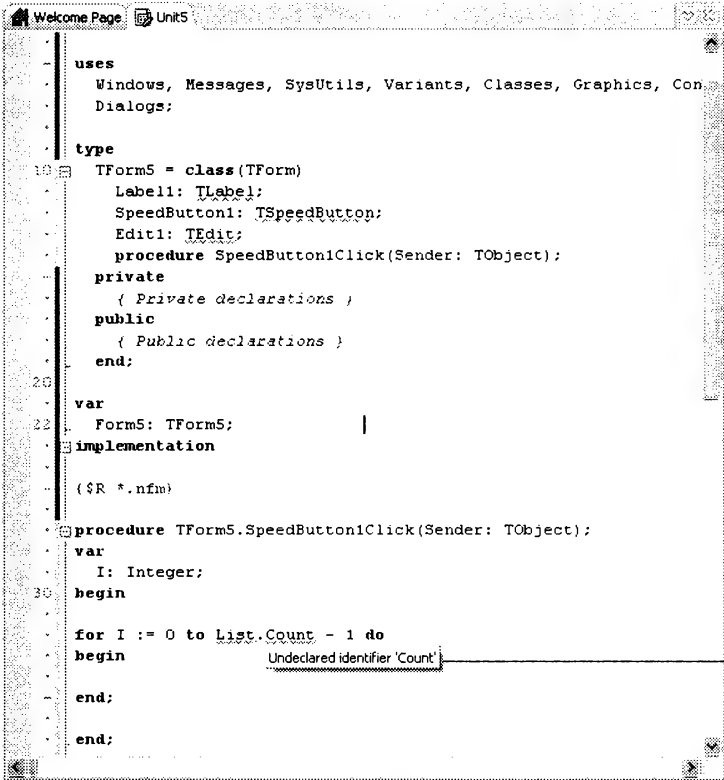


Рис. 2.2. Проверка исходного текста в режиме реального времени



Чтобы закрыть все окна, открытые в редакторе, кроме текущего, щелкните правой кнопкой мыши на названии вкладки для текущего модуля и в открывшемся контекстном меню выберите команду *Close All Other Pages* (Заккрыть все другие страницы).

Далее мы рассмотрим важнейшие особенности редактирующей системы Delphi.

2.2. Визуализация кода

Линейка в левой части окна редактора содержит номера строк (с шагом по 10 строк), что хорошо помогает ориентироваться в коде, где немало свернутых фрагментов. Кроме того, точный номер показан для строки, в которой в данный момент находится курсор. Строки, которые были модифицированы после последнего сохранения текущего файла, выделяются желтым цветом, а строки, модифицированные в текущей сессии, — зеленым.

Новое окно Просмотрщика структуры (Structure View) позволяет работать с исходным текстом на более высоком логическом уровне. Просмотрщик визуально представляет классы проекта и используемые модули. Щелчком мыши можно быстро переместиться к соответствующему определению в исходном тексте. Имеется также раздел Ошибки (Errors) с перечнем выявленных синтаксических ошибок.

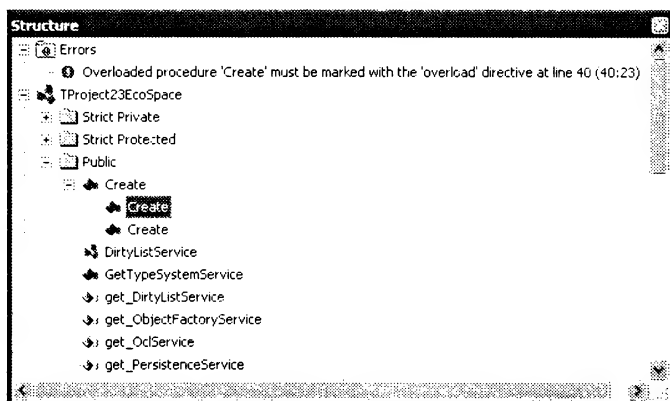


Рис. 2.3. Средство просмотра объектной структуры кода

2.3. Построение модели кода (Model View)

Важнейшая особенность Delphi заключается в возможности представления кода диаграммами классов UML. Если при этом в исходный код вносятся изменения, происходит автоматическое обновление модели. Отметим,

что данную возможность не следует путать с поддержкой в Delphi режима UML-моделирования с помощью встроенной системы Borland Together и ЕСО-моделей. Визуализация кода просто ускоряет процесс перемещения по исходному тексту, а концепция моделирования предполагает более развитые механизмы проектирования всей логики системы на уровне моделей.

Для перехода от Менеджера проектов (фактически, от обычной структуры файлов проекта) к модельному представлению надо дать команду View ➤ Model View (Просмотр ➤ В виде модели). К этому же окну можно перейти и через вкладку Model View окна в правом верхнем углу, где совмещены Менеджер проектов, Средство модельного представления и Проводник данных.

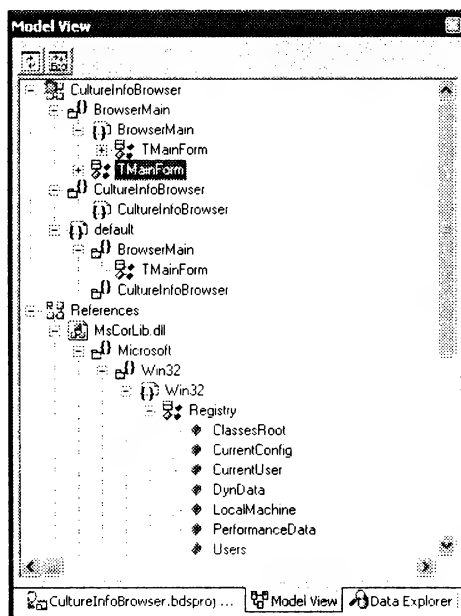


Рис. 2.4. Средство модельного представления проекта

Для перехода к определению того или иного элемента в структуре моделей (описанию класса и любых его элементов) достаточно дважды щелкнуть мышью на соответствующей строке структуры. А находясь в редакторе кода, можно выбрать любой из элементов описания и дать команду контекстного меню Synchronize Model View (Синхронизировать с модельным представлением) — в окне Model View сразу же выделится соответствующий элемент.

Если необходимо перейти от модельного представления к UML-диаграммам, надо выбрать нужный элемент и в контекстном меню дать команду Select on Diagram (Выбрать на диаграмме). Если текущий класс на диаграмме отсутствовал, он добавится к диаграмме автоматически, если уже присутствовал — выделится его нужная часть.

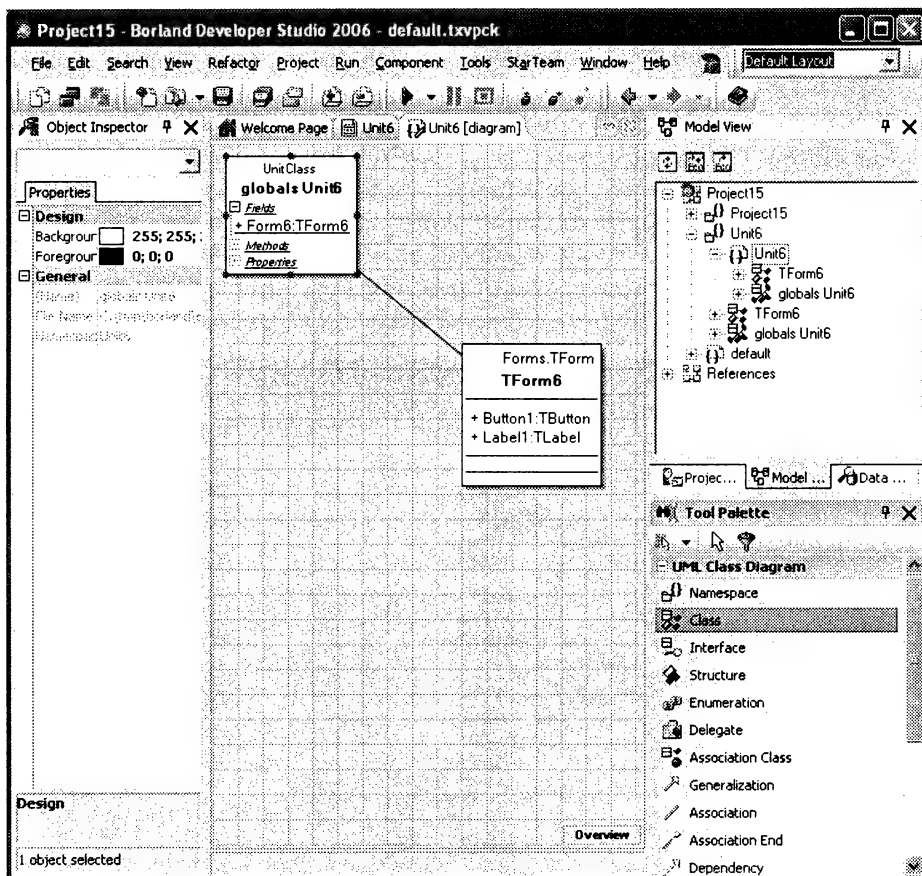


Рис. 2.5. Пример модельного представления

На каждой диаграмме указывается имя представляемого класса, его поля и методы. Если в проекте несколько классов, все они будут размещены на одной схеме (рис. 2.5).

2.4. Подсказки в коде (Code Insight)

Режим отображения подсказок в коде включается через окно настройки среды: Tools > Options > Code Insight. Он существует в системе Delphi на протяжении уже нескольких версий, постоянно дополняется и развивается.

Средства настройки параметров режима автоматической подсказки расположены на панели Automatic Features. Они означают следующее:

- Code completion (Автозавершение) — предлагается название свойства, типа, метода или события в списке после указания названия объекта или имени класса и последующей точки;

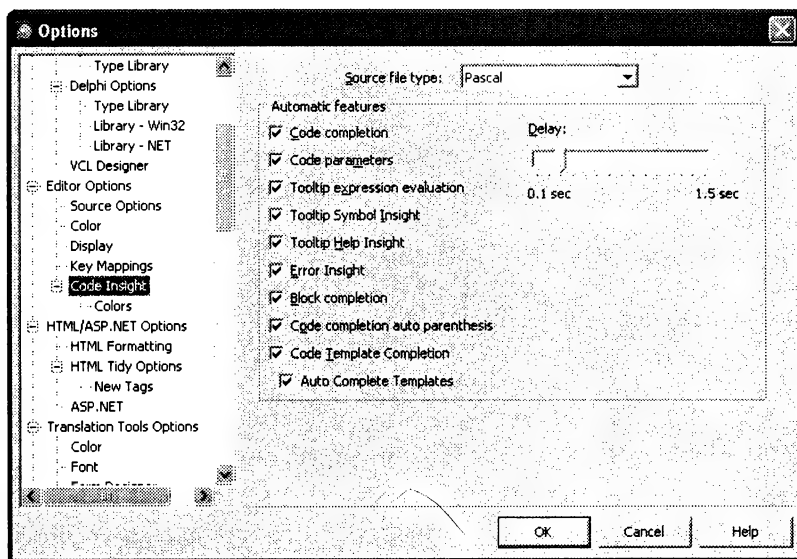


Рис. 2.6. Настройка режима автоматической подсказки

- **Code parameters (Автозаполнение)** — предложение параметров процедуры или метода после ввода соответствующего названия и открывающей скобки;
- **Tooltip expression evaluation (Автовычисление)** — подсказка с текущим значением переменной или выражения при наведении курсора на соответствующий идентификатор, всплывающая в процессе пошаговой отладки;
- **Tooltip Symbol Insight (Автоописание)** — всплывающая подсказка с описанием идентификатора при наведении на него курсора;
- **Tooltip Help Insight (Автосправка)** — дополняет предыдущий пункт ссылкой на тематический раздел;
- **Error Insight (Автокоррекция)** — выделение потенциально ошибочных мест в исходном тексте до компиляции, в процессе набора текста. Такие места подчеркиваются красным, а при наведении на них курсора всплывает подсказка с кратким описанием проблемы.



Разным типам файлов с исходными текстами (например, Паскаля или Си++) соответствуют разные типы настроек-подсказок. Тип языка программирования выбирается в раскрывающемся списке **Source file type**.

Как быстро появится окно подсказки, определяет параметр **Delay** (в секундах). Цвета различных элементов подсказок задаются в подразделе **Colors**.

2.5. Шаблоны кода (Code Templates)

Шаблоны типовых конструкций и операторов языка Delphi можно вводить в текущую позицию курсора одним щелчком мыши. Это выполняется с помощью так называемого Менеджера шаблонов, который вызывается командой View ➤ Templates (Вид ➤ Шаблоны), когда в главном окне открыт редактор исходных текстов.

После вставки шаблона в исходный текст его код подсвечивается светлым цветом, а ряд его ключевых элементов выделяется синим шрифтом с подчеркиванием. По ним можно быстро перемещаться нажатием клавиши Tab (или Shift + Tab в обратном порядке). Например, выбрав оператор цикла (рис. 2.8), мы можем сначала откорректировать имя переменной-счетчика I, затем, нажав клавишу Tab, автоматически перейдем к редактированию начального значения цикла (0), а потом — конечного List.Count-1:

```
for I := 0 to List.Count-1 do
```

Наконец, нажав клавишу Tab в последний раз, попадем на следующую строчку за оператором for, в позицию ввода тела цикла. При этом подсветка снимется, и далее редактирование продолжится в обычном режиме.

Более того, данный режим функционирует и в автоматическом режиме. При попытке ввода текста, соответствующего ключевому элементу одного из шаблонов, тот автоматически вызывается в текущее место. Так, набрав for и нажав пробел, мы сразу вызовем соответствующий шаблон.

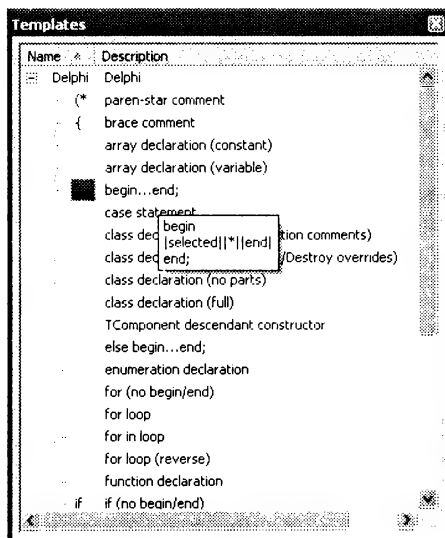


Рис. 2.7. Менеджер шаблонов в действии

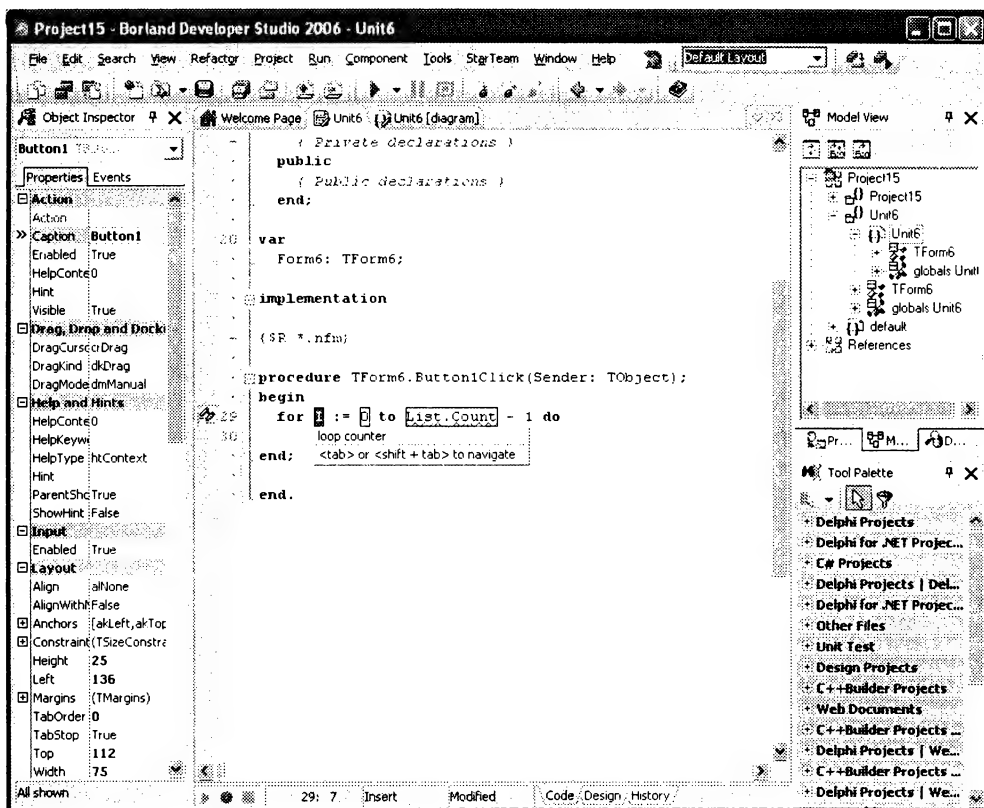


Рис. 2.8. Автоматическое применение шаблонов кода



В некоторых случаях шаблон даже автоматически вставляет описание переменной-счетчика в качестве локальной переменной текущего метода.

Список шаблонов кода (они еще называются «кусочками», snippets) можно легко расширить самому, выделив в исходном тексте некоторый блок, затем зажав клавишу ALT и перетаскив выделенный блок мышью на палитру инструментов. Новый «кусочек» добавится в конец раздела и получит название по первым символам блока кода. Впоследствии удалить его можно через контекстное меню.

Кроме того, с помощью менеджера шаблонов можно выполнить более тонкую настройку. При выборе какой-либо строки в менеджере в основном окне редактора появится детальное описание этого шаблона, выполненное в XML-формате, а в небольшом дополнительном окне самого менеджера — краткое описание шаблона. В этом формате можно запрограммировать очень сложные текстовые шаблоны (рис. 2.9).

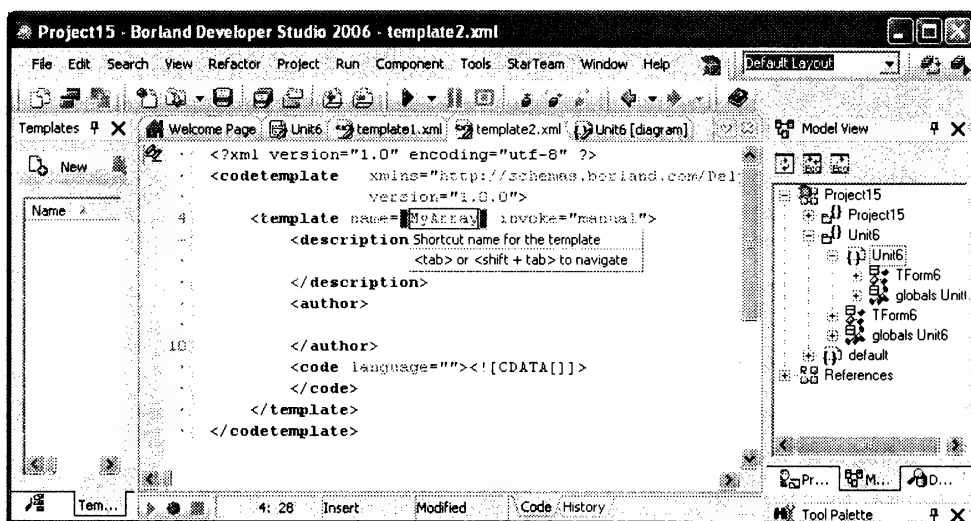


Рис. 2.9. Описание шаблона в формате XML

Различные части кода можно также «обертывать» подходящими языковыми средствами, например, быстро заключить в логический блок, разместить внутри оператора цикла или *try*-блока и так далее. Для этого надо выделить в редакторе нужный набор команд, вызвать его контекстное меню и дать команду Surround, выбрав в подменю нужный оператор Delphi.

2.6. Умные блоки (Smart Block)

Система Delphi отслеживает парность введенных логических скобок *begin/end*. В частности, набрав в редакторе ключевое слово *begin*, достаточно нажать клавишу Enter, чтобы среда автоматически добавила завершающую скобку *end*.

К этой же функции относится и режим быстрого завершения описания класса. Допустим, имеется код класса, в котором описано множество методов, однако некоторые (или все) пока не имеют реальных прототипов в разделе реализации. Ранее в лучшем случае можно было установить курсор на отдельный метод и нажать комбинацию Ctrl + Shift + C, в результате чего в код автоматически добавлялась пустая реализация этого метода. Delphi 2006 выполняет это действие для всех методов сразу, создавая все необходимые прототипы процедур и функций.

2.7. Средства компактного свертывания кода (Code Folding)

Разные части кода можно сворачивать (схлопывать) и разворачивать, акцентируя тем самым внимание на наиболее важных в данный момент задачах

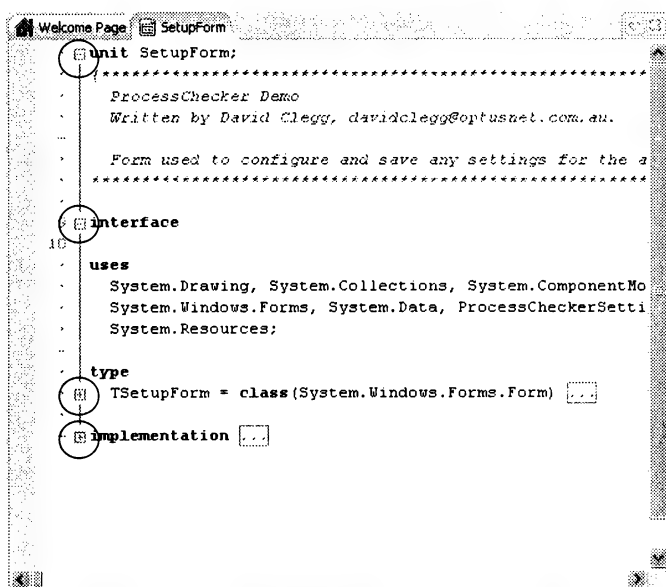


Рис. 2.10. Узлы сворачивания кода

кодирования. Для этого достаточно щелкнуть мышью на небольших значках «+» и «-», расположенных на линейке с номерами строк в левой части редактора (рис. 2.10).

Свернутый код помечается в конце соответствующей строчки либо тремя точками в прямоугольнике, либо текстовым названием свернутого блока в прямоугольнике, например так, как показано на рисунке 2.11 (раздел Designer Managed Code).

Чтобы дать название сворачиваемой части кода, ее надо поместить в конструкцию препроцессора:

```
{ $ REGION 'название-части' }
```

сворачиваемый код:

```
{ $ENDREGION }
```

Сворачивать можно описания классов, разделы интерфейса и реализации, тексты процедур, функций и методов. Можно также вкладывать сворачиваемые части друг в друга.



Удобная возможность быстрого перехода между реализациями методов — к следующему или к предыдущему, возможна комбинациями клавиш **Ctrl + Alt + стрелка-вверх** или **Ctrl + Alt + стрелка-вниз** соответственно. Также возможен ускоренный переход к самому первому и самому последнему методам в разделе реализации комбинациями **Ctrl + Alt + PgUp** и **Ctrl + Alt + PgDn**.

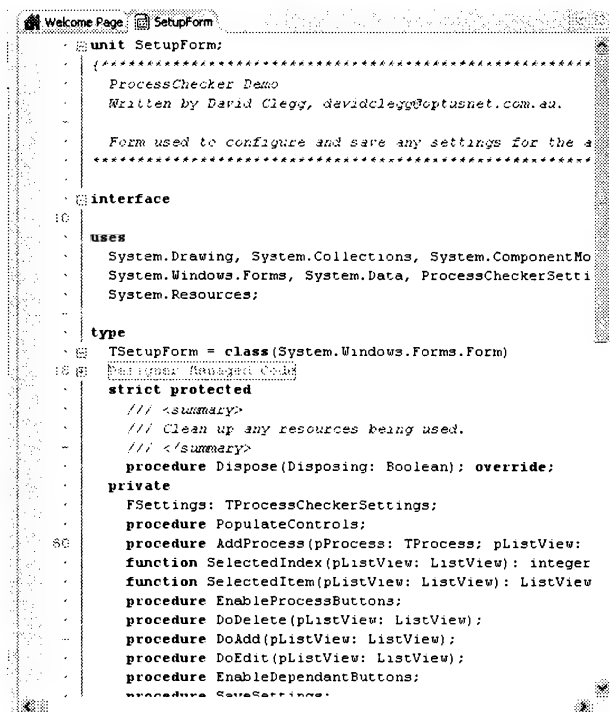


Рис. 2.11. Свернутый блок кода

2.8. Организация закладок

Классическая возможность Delphi позволяет задать до 10 закладок в каждом файле, открытом в окне редактора. Закладка ставится комбинацией клавиш **Ctrl + K + цифра** от 0 до 9. В дальнейшем быстро переместиться к закладке можно, нажав комбинацию **Ctrl + соответствующая цифра**. Наличие закладки в некоторой строке индицируется небольшим зеленым прямоугольником в левой части редактора. Что особенно приятно, теперь закладки сохраняются при закрытии проекта и после повторной загрузки восстанавливаются автоматически — ранее они пропадали. Для явного их удаления надо воспользоваться командой **Clear Bookmarks (Очистить закладки)** контекстного меню редактора.

2.9. Синхронное редактирование (Code Sync)

Режим синхронного редактирования (Sync Edit) позволяет одновременно править все одинаковые части кода. Для этого надо выделить в исходном тексте некоторый блок и нажать значок в форме бинокля слева от линейки с номерами строк — в результате все одинаковые идентификаторы будут выделены (рис. 2.12).

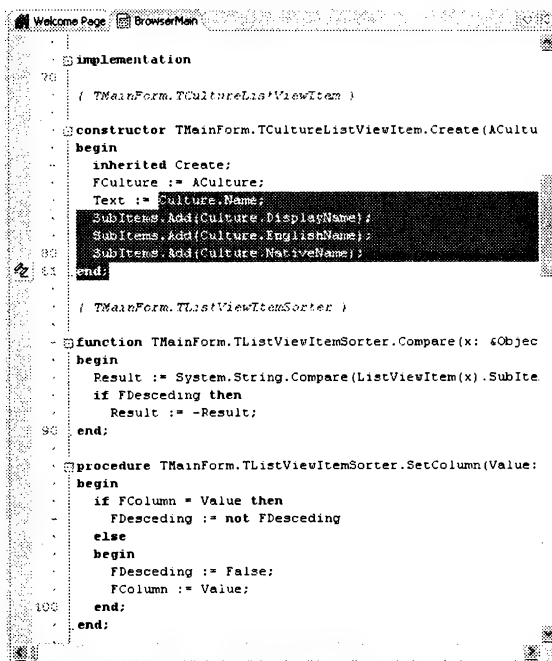


Рис. 2.12а. Выделяем блок текста...

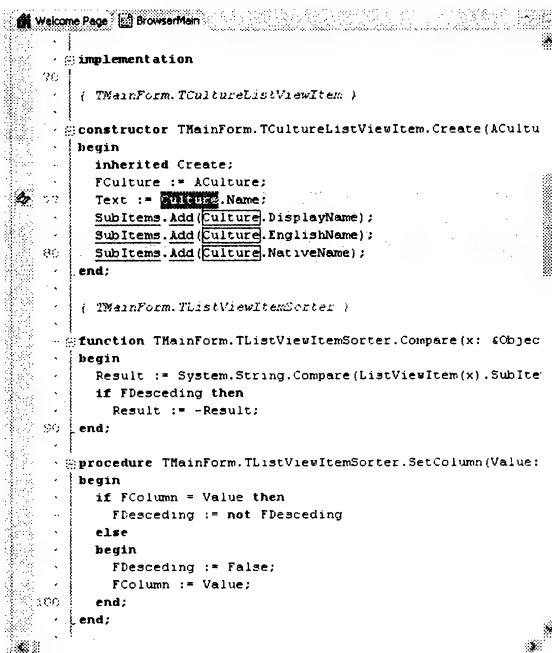


Рис. 2.12б. ...и начинаем синхронную правку кода

Курсор находится у самого первого идентификатора, и при его редактировании одновременно модифицируются и все другие экземпляры данного идентификатора в выделенном блоке. Далее можно перейти к следующей выделенной группе одинаковых идентификаторов. Для выхода из режима синхронного редактирования нажмите клавишу Esc.

2.10. Ведение списка отложенных дел (ToDo)

Отметим такую крайне полезную возможность Delphi, как ведение списка отложенных задач (ToDo List). С его помощью программист оставляет по всему коду своеобразные заметки — напоминающие сообщения, которые в будущем помогут быстро сориентироваться в незаконченных модулях кода.

Заметки удобно создавать, находясь в редакторе. Разместив курсор в строке кода, куда надо добавить напоминание (оно будет представлено в исходном тексте в виде обычного комментария, отличающегося специальным форматом), в контекстном меню выберите пункт Add To-Do List Item (Добавить элемент списка напоминаний), после чего появится диалоговое окно формирования такого напоминания (рис. 2.13).

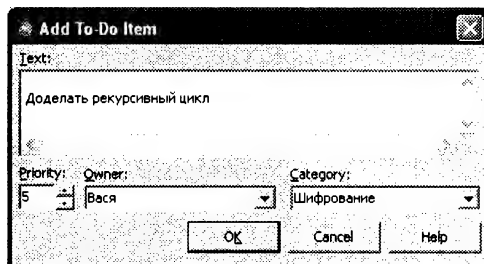


Рис. 2.13. Добавление напоминания

В поле Text вводится описание напоминания, в поле Priority — приоритет, важность (разрешены значения от 0 до 5), в поле Owner (Владелец) — имя автора напоминания (вводится в произвольной форме или выбирается из уже существующих), в поле Category (Категория) — тематическая группа напоминания (заполняется так же, как и поле Owner).

После нажатия кнопки ОК в исходный текст добавится следующая строка:

```
{ TODO 5 -oВася -сШифрование :  
Доделать рекурсивный цикл }
```

Модифицировать эту строку, конечно, нельзя, иначе потом система не сможет распознать такой комментарий как элемент списка напоминаний.

Впоследствии, когда деятельность, связанная с подобной заметкой, будет завершена, соответствующий пункт списка напоминаний надо пометить как

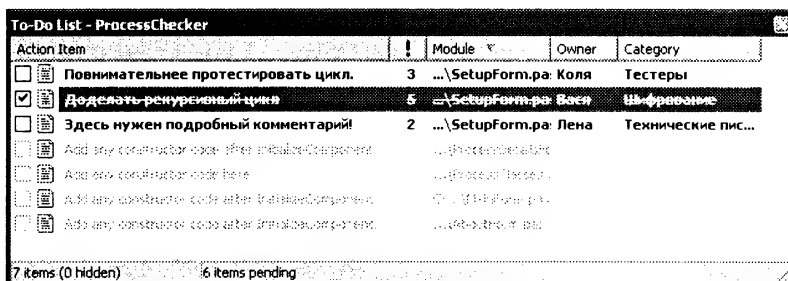


Рис. 2.14. Отметка об устранении неподелок

«завершенный». Для этого достаточно вызвать основное диалоговое окно обработки списка напоминаний: View > To-Do List (Просмотр > Список напоминаний), и в нем установить слева флажок. При этом строка с текстом напоминания будет записана зачеркнутым шрифтом (рис. 2.14).

Само окно списка напоминаний содержит удобные средства быстрого поиска, сортировки и фильтрации подсказок. Сортировка происходит просто нажатием на заголовке соответствующей колонки, а фильтрация — вызовом из контекстного меню данного окна дополнительного диалогового окна Filter (Фильтр списка напоминаний), в котором можно указывать различные выбранные разделы (например, категории или пользователей).

И конечно, любую строку списка напоминаний можно в дальнейшем как отредактировать, или удалить, просто выделив ее в списке и выбрав в контекстном меню пункт Delete (Удалить). Но применять этот прием рекомендуется только в крайних случаях (например, при вводе ошибочной заметки), так как гораздо полезнее полностью сохранять всю историю сделанной работы.

2.11. Быстрое комментирование

Любую часть кода можно быстро превратить в комментарий. Для этого его надо выделить, после чего нажать комбинацию клавиш Ctrl + /, и в начало каждой выделенной строки добавятся символы комментария //.

2.12. Расширенные комментарии

В дополнение к однострочным комментариям в стиле Си `"/ /"`, уже давно существующим в Delphi, в новой версии появился их расширенный вариант, состоящий из трех символов: `"/ / /"`. Он применяется для указания компилятору, что текст, следующий за ним, должен быть использован системами автоматического ведения документации (формирования XML-документа). Эта технология исходно реализована в системе Microsoft Visual C#. Фактически, такой трехсимвольный комментатор допускает указание набо-

ра XML-тегов, на основе которых и будет генерироваться эта документация, в итоговом виде представляемая в виде XML-файла. Список рекомендуемых тегов можно найти в документации .NET SDK, в разделе «Recommended Tags for Documentation Comments».

Как получить результирующий документ? Надо установить соответствующий флажок в окне параметров проекта (рис. 2.15): Project > Options > Compiler > Documentation > Generate XML documentation (Проект > Параметры > Компилятор > Документирование > Генерировать документацию XML). Теперь после компиляции проекта в его рабочем каталоге будут созданы файлы: имя-проекта.xml и имя-модуля.xml.

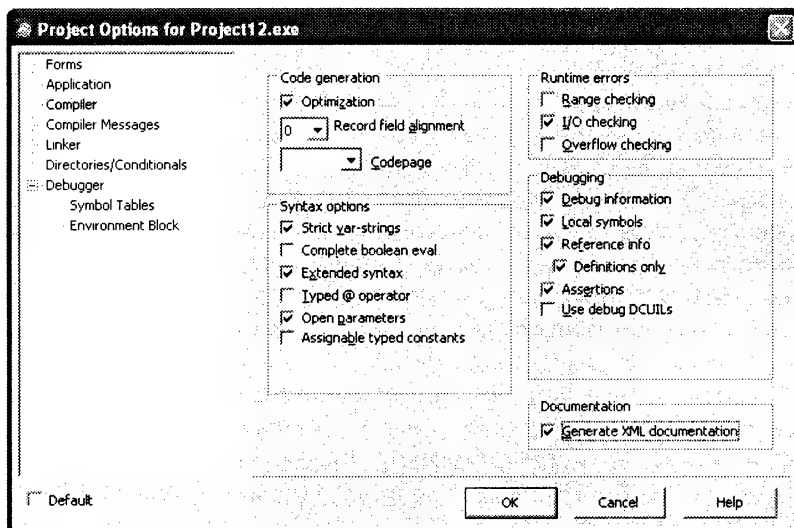


Рис. 2.15. Настройка автоматической генерации документации XML

2.13. Клавиатурные макросы

Некоторые часто повторяемые последовательности нажатий на клавиши (например, для доступа к глубоко расположенным пунктам строки меню) можно записать в виде макропоследовательностей.

Сделать это просто. Достаточно нажать красную кнопку Record Macro (Запись) в нижней части окна редактора, после чего выполнить нужные действия с клавиатуры, а затем завершить запись нажатием на кнопку Stop Recording Macro (Стоп). Для воспроизведения макроса надо разместить курсор в нужной позиции редактора и нажать кнопку Playback Macro (Воспроизведение). Тотчас повторится ранее указанная последовательность нажатий клавиш.

Отметим, что макрос хранится лишь на протяжении текущей сессии работы. После перезапуска среды он теряется.

Глава 3. Технологии интеллектуального редактирования (рефакторинга)

Средства рефакторинга, безусловно, являются одними из наиболее сильных нововведений последних версий Delphi. Несмотря на схожесть с механическими операциями редактирования, операции рефакторинга вносят в код гораздо более глубокие, смысловые изменения. Ведь многие из них невозможно выполнить, не понимая программного синтаксиса и взаимосвязи описаний классов.

3.1. Поиск модуля (Find Unit)

Данная возможность, вызываемая обычно из контекстного меню редактора: Refactor ➤ Find Unit (Рефакторинг ➤ Найти модуль), позволяет добавить к списку подключенных модулей (в ключевом слове `uses`) какой-либо из стандартных модулей, отвечающих строке поиска. Точка его включения (в раздел интерфейса или в раздел реализации) задается соответствующим переключателем в нижней части окна (рис. 3.1).

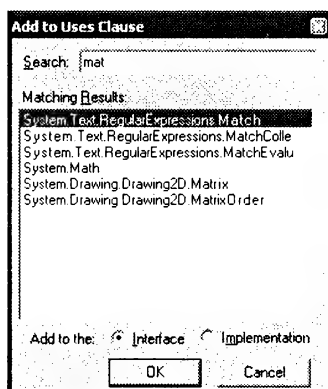
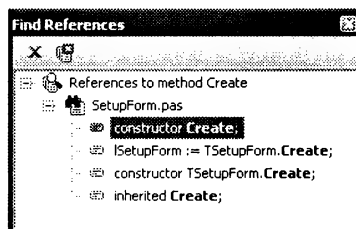


Рис. 3.1. Поиск и подключение модуля

3.2. Поиск ссылок (Find References)

Данная возможность позволяет выявить наличие некоторого идентификатора во всех связанных с ним файлах проекта. Идентификатор надо выделить в редакторе, после чего дать команду **Search > Find References** (Поиск > Найти ссылки). В появившемся окне открывается структура с результатами поиска.



При перемещении по результирующей структуре в редакторе синхронно будет отображаться нужная строка соответствующего файла. Для удобства узлы дерева можно безболезненно удалять — исходного кода это не коснется.

Для поиска локальных идентификаторов можно воспользоваться вариантом данной команды **Find Local References** (Найти локальные ссылки), а если надо ограничиться лишь текущим файлом с исходным текстом, надо дать команду **Find Declaration Symbol**.

3.3. Поиск класса (Find Class)

Find Class — новая возможность среды Delphi по поиску класса. Она вызывается командой **Search > Find Class** (Поиск > Найти класс) и позволяет быстро найти нужный класс, реализованный в библиотеках Delphi, .NET или Win32. Причем поиск происходит сопоставлением не с первым, а с последним элементом каждого пространства имен. То есть, если нам надо найти класс **TWinForm7**, созданный в рамках текущего проекта, то, набрав уже первые символы его названия, мы получим ссылку на класс **WinForm7.TWinForm7** (рис. 3.2).

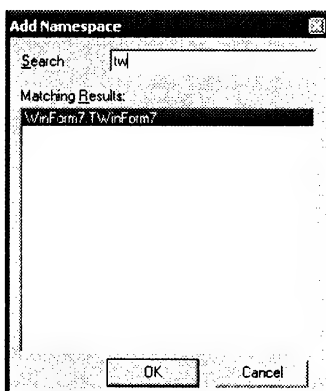


Рис. 3.2. Пример поиска класса по первым символам его имени

После нажатия на кнопку **OK** редактор переместит курсор к определению найденного класса.

3.4. Быстрая декларация переменной и поля (Declare Variable, Declare New Field)

Данная возможность доступна, когда по результатам работы компилятора выявились необъявленные идентификаторы. На такой идентификатор надо установить курсор, дать команду Refactor ➤ Declare Variable (Рефактор ➤ Объявить переменную), в диалоговом окне Declare New Variable (Объявить новую переменную) указать идентификатор, его тип, начальное значение и при необходимости процедуру, в которой эту переменную надо объявить (рис. 3.3).

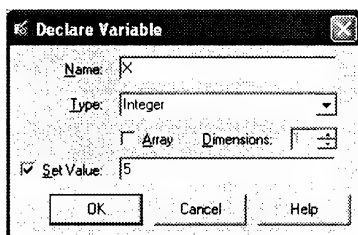


Рис. 3.3. Объявление переменной

Удобно пользоваться этим средством, когда код дополняется новыми функциями и требуется быстро задействовать дополнительные локальные переменные, например, для выполнения новых циклов.

Схожая возможность доступна и для имени поля класса. Если некоторый неопределенный идентификатор должен считаться в коде не локальной переменной, а полем класса, надо дать команду Refactor ➤ Declare New Field (Рефактор ➤ Объявить новое поле), в диалоговом окне дополнительно будет предложен соответствующий класс, где поле должно быть определено, а также его область видимости.

3.5. Визуальное создание суперкласса (Extract Superclass)

На базе существующего класса можно быстро создать его «предок» — класс, включающий подмножество свойств и методов текущего класса. Для этого надо выделить конкретный класс в средстве просмотра модели: View ➤ Model View (Вид ➤ В виде модели), или в окне диаграммы классов, если таковая создана, и дать команду Refactor ➤ Extract Superclass (Рефакторинг ➤ Извлечь суперкласс). Появится диалоговое окно, в котором можно указать члены исходного класса, подлежащие переносу, а также дополнительно уточнить, будет ли тот или иной метод абстрактным (рис. 3.4).

Схожим способом извлекаются и интерфейсы: выделением класса и последующей командой Refactor ➤ Extract Interface (Рефакторинг ➤ Извлечь интерфейс).

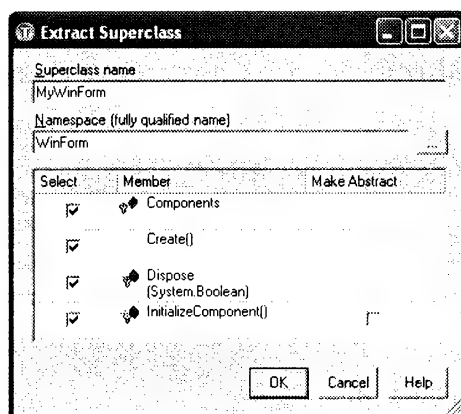


Рис. 3.4. Формирование суперкласса и списка включаемых методов

3.6. Визуальное создание метода (Extract Method)

Разработчик может выделить произвольную часть исходного кода и применить к ней операцию рефакторинга Extract Method (Преобразовать в метод) — рис. 3.5.

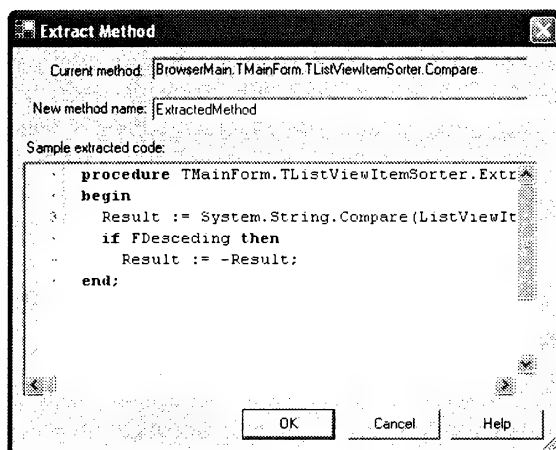


Рис. 3.5. Формирование метода на основе фрагмента кода

В результате система попытается создать на базе этого кода новый метод (его название указывается в поле New method name), и впоследствии к соответствующим операторам выделенного блока можно будет обращаться как к этому новому методу. При этом в процессе рефакторинга автоматически сформируются параметры метода и его локальные переменные. Выделенная часть кода должна быть, конечно, достаточно целостной по реализуемой логике.

3.7. Визуальное изменение списка параметров процедуры (Change Parameters)

Выделив имя процедуры, можно в визуальном редакторе откорректировать ее параметры, указав их название, тип и начальное значение. Это особенно полезно, если число параметров велико, а некоторые из них описаны как имеющие значения, заданные по умолчанию.

Вызов редактора параметров происходит после выделения названия процедуры и выполнения команды контекстного меню **Refactor** ➤ **Change Parameters** (Рефакторинг ➤ Изменить параметры).

3.8. Глобальное переименование идентификатора (Rename)

Операция глобального переименования идентификатора во всех файлах проекта применима к названиям переменных, методов, полей, классов, структур, интерфейсов и параметров. При этом накладываются естественные ограничения на процесс переименования: так, нельзя переименовывать переменную в ключевое слово.

Соответствующая команда **Refactor** ➤ **Rename** (Рефакторинг ➤ Переименовать) вызывает диалоговое окно, которое позволяет ввести новое название идентификатора в поле **New Name**. Можно запросить предварительный просмотр дерева вхождений этого идентификатора в код, включив флажок **View references before refactoring** (Просмотр ссылок перед рефакторингом), в противном случае сразу после нажатия кнопки **OK** будет выполнено изменение названия идентификатора во всем проекте. Установка флажка позволяет исключить некоторые экземпляры идентификаторов из операций нажатием на кнопку **Remove Refactoring** (Удалить из рефакторинга). При этом, конечно, в исходном тексте сам идентификатор остается, просто он не включается в процесс модификации. А выполнение рефакторинга выполняется нажатием кнопки **Refactor** (рис. 3.6).

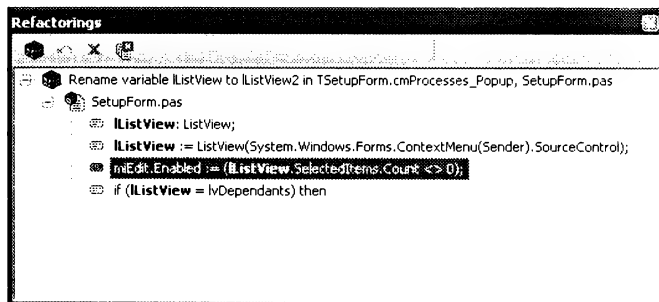


Рис. 3.6. Подготовка к переименованию идентификатора

3.9. Перемещение поля внутрь класса (Introduce Field/Variable)

Удобное средство быстрого создания нового элемента класса непосредственно из тела какого-либо метода позволяет вести кодирование, не перемещаясь многократно к описанию класса и обратно к операторам, реализующим его методы. Для этого сначала надо выделить какое-либо выражение в коде. Например, в таком обработчике:

```
procedure TWinForm.Button1_Click(sender: System.Object;  
e: System.EventArgs);  
  var x,y: Integer;  
  begin  
    x := 10;  
    y := x*2 + 5;  
  end;
```

мы хотим взять выражение `x*2 + 5` и присвоить его новому полю класса `TWinForm`. Для этого выражение выделяется в редакторе и дается команда **Refactor** ➤ **Introduce Field** (Рефакторинг ➤ Создать поля), в диалоговом окне указывается название нового поля (**Name**), будет ли оно статичным (флажок **Static**) и его видимость (раскрывающийся список **Visibility** со спецификаторами **public**, **private** и тому подобными). Если, допустим, указать название нового поля `MyFld`, то после нажатия на кнопку **OK** код изменится так:

```
procedure TWinForm.Button1_Click(sender: System.Object;  
e: System.EventArgs);  
  var x,y: Integer;  
  begin  
    x := 10;  
    Self.MyFld := x*2 + 5;  
    y := MyFld;  
  end;
```

А в описание класса `TWinForm` добавится новое поле:

```
public  
var  
  MyFld: integer;
```

Если вместо команды **Refactor** ➤ **Introduce Field** (Рефакторинг ➤ Создать поле) дать команду **Refactor** ➤ **Introduce Variable** (Рефакторинг ➤ Создать переменную), то аналогично будет создана локальная переменная в пределах ближайшей процедуры или метода.

3.10. Перемещение описания между классами (Move)

Этот мощный и ресурсоемкий прием позволяет переместить статический член класса в другой класс, или сам класс — в другое пространство имен. Для этого курсор устанавливается на соответствующем идентификаторе и дается команда Refactor > Move (Рефакторинг > Переместить), после чего в диалоговом окне указывается (выбирается в дереве) либо новое пространство имен, либо нужный целевой класс.

3.11. Перемещение членов между классами (Pull Members Up/Down)

Члены классов можно быстро перемещать либо от дочернего класса к одному из родительских (команда Refactor > Pull Members Up), либо от родительского к одному из классов-наследников (команда Refactor > Pull Members Down). Если команда применима к выделенному в исходном коде элементу класса, то открывается диалоговое окно, в котором предлагается выбрать итоговый класс, куда будут добавлены выбранные элементы (рис. 3.7).

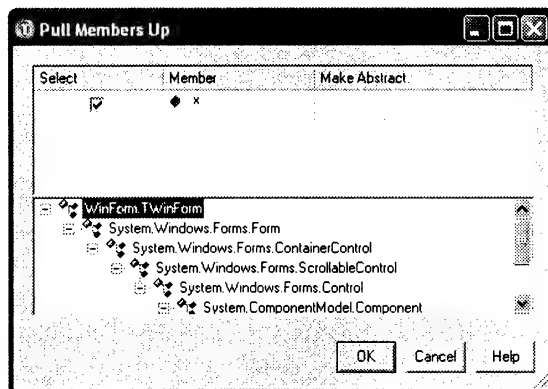


Рис. 3.7. Выбор перемещаемых членов класса

При этом из описания исходного класса соответствующие элементы, конечно, удаляются, иначе данный прием не имел бы особого смысла.

3.12. Удаление лишних переменных (Inline Variable)

С помощью данной возможности можно выполнить задачу, обратную предыдущей: удалить из исходного текста упоминание о некоторой переменной, если она используется лишь временно в качестве хранилища промежуточного значения. Если у нас есть код:


```
procedure TwinForm.Button1_Click(sender: System.Object;  
e: System.EventArgs);  
var  
xxx: Integer;  
x,y: Integer;  
begin  
  x := 10;  
  xxx := x*2 + 5;  
  Self.MyFld := xxx;  
end;
```

то можно выделить первое упоминание переменной `xxx`, где ей присваивается начальное выражение `x*2 + 5`, и дать команду **Refactor** ➤ **Inline Variable** (Рефакторинг ➤ Скрыть переменную). Код в результате изменится на такой.

```
begin  
  x := 10;  
  ;  
  Self.MyFld := x*2 + 5;  
end;
```

Переменная `xxx` из кода была автоматически удалена, а записываемое в нее значение (выражение `x*2 + 5`) переместилось непосредственно в следующий оператор.

3.13. Безопасное удаление свойств и методов (Safe Delete)

Безопасное удаление свойств и методов классов выполняется выделением соответствующего элемента и выполнением команды **Refactor** ➤ **Safe Delete** (Рефакторинг ➤ Удалить). В диалоговом окне представляется список классов, содержащих данное свойство или метод, после чего нажатием на кнопку ОК выполняется автоматическое удаление элемента из всех классов.

3.14. Быстрое перемещение строки в ресурсы (Extract Resource String)

Удобная возможность компилятора Delphi по размещению всех строковых констант в разделе ресурсов результирующего кода значительно упрощающая процесс локализации приложения, не всегда пользуется популярностью. С помощью команды рефакторинга **Extract Resource String** (Преобразовать строковую константу в ресурс) в программе создается раздел `resourcestring` (если он не существовал ранее), и выбранная строка записывается в него с заменой в исходном тексте на идентификатор, указанный в поле **Name**.

Так, если в коде:

```
xx := 'Мама';
```

установить курсор на строке 'Мама' и дать команду Refactor ➤ Extract Resource String (Рефакторинг ➤ Преобразовать строку в ресурс), то откроется диалоговое окно, в котором надо указать подходящее название строковой константы в ресурсах (например, StrMama) — рис. 3.8.

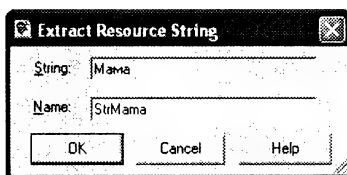


Рис. 3.8. Преобразование строковых данных в ресурс

После нажатия на кнопку ОК код изменится на:

```
xx := StrMama;
```

А в начало раздела реализации добавится такой код:

```
resourcestring  
    StrMama = 'Мама';
```

3.15. Отмена операций рефакторинга

Особенность всех операций рефакторинга: перед их выполнением можно оценить результат в режиме предварительного просмотра. Если в процессе длительной операции рефакторинга встретятся ошибки, то все промежуточные модификации будут ликвидированы и код вернется в исходное состояние.

Операцию рефакторинга обычно отменяют стандартной командой меню View ➤ Undo (Просмотр ➤ Отменить, Ctrl + Z), однако для некоторых операций придется применять специализированный вариант этой команды. Более подробно выполненные действия рефакторинга отображаются в окне Refactorings, вызываемого из главного меню командой View ➤ Refactorings (Просмотр ➤ Рефакторинги).

Глава 4. Технологии компиляции и отладки

В связи с тем, что система Delphi, начиная с 8-й версии, пополнилась средствами широкой поддержки платформы .NET, в язык программирования Delphi был внесен ряд дополнений. Связаны они прежде всего с возникшим пересечением названий ряда ключевых слов Delphi с идентификаторами, применяемыми в библиотеках классов .NET Framework.

Мы рассматриваем все изменения, происшедшие в языке Delphi, начиная с версии Delphi 8, в которой впервые появилась поддержка технологии .NET.

4.1. Новое в языке

4.1.1. Использование расширенного набора символов

За счет поддержки универсальной схемы кодирования Unicode в программе на языке Delphi стало возможным применять не только символы ASCII, но и символы русского языка. В частности, отныне названия переменных и функций можно записывать кириллицей:

```
var Хомяк, Котенок: Integer;  
begin  
    Хомяк := 4;  
    Котенок := Хомяк - 5;  
end;
```

Отношение к такой возможности у разработчиков неоднозначное, и если начинающим программистам и создателям приложений для личного пользования вполне подойдет использование переменных с названиями наподобие «Ы» (верхний и нижний регистры в языках Delphi и Паскаль традиционно не различаются), то при работе в коллективе желательно все же придерживаться латинского алфавита, а кириллицу применять лишь в комментариях. При этом надо учитывать такое ограничение Delphi, как недопустимость записи символами Unicode названий идентификаторов в public-разделах описаний классов (как в качестве названий полей, так и в качестве названий типов).

4.1.2. Поддержка пространства имен

Одна из фундаментальных концепций технологии .NET, физически поддерживаемая средой выполнения CLR, — так называемое пространство имен: иерархическая система организации видимости идентификаторов (названий типов, переменных, функций). В системе Delphi эта концепция реализована с помощью модулей (*units*), когда программист размещает в интерфейсном разделе набор доступных для других модулей идентификаторов, обработка которых скрыта в разделе реализации. Нередко в прикладной программе по каким-либо причинам необходимо использовать уже существующий в библиотеках Delphi идентификатор в виде, например, собственной переменной. В соответствии с фундаментальной семантикой Паскаля, любой идентификатор оценивается компилятором прежде всего с точки зрения его локальности, и лишь потом, если не найдено его описание в пределах текущей подпрограммы (в виде локальной переменной или в виде параметра), поиск продолжается в пределах текущего класса, модуля и так далее. Поэтому, если в некотором операторе надо применить стандартный идентификатор какой-либо библиотеки Delphi, а он уже задействован в текущем модуле в качестве, например, локальной переменной, то перед этим стандартным идентификатором, отличающимся от локального описания, надо дополнительно через точку указать префикс: имя модуля, в интерфейсной части которого он описан. Следующий код ошибочен:

```
var IntToStr: String;  
begin  
    IntToStr := '5';  
    IntToStr := IntToStr(10);  
end;
```

Идентификатор `IntToStr` используется и как локальная переменная, и как стандартная функция преобразования числа в строку, но так как идентификатор в первую очередь рассматривается на уровне локальных определений, то выражение `IntToStr(10)` считается ошибочным. Чтобы вызов работал корректно, укажем перед ним название стандартного модуля `SysUtils`:

```
var IntToStr: String;  
begin  
    IntToStr := '5';  
    IntToStr := SysUtils.IntToStr(10);  
end;
```

С появлением большого числа новых классов .NET (несколько тысяч) ситуация с поддержкой соответствующих идентификаторов усложнилась тем, что подход Delphi ориентирован на одноуровневое представление (модуль-префикс и идентификатор), а классы .NET организованы в виде объемной иерархии, и к различным идентификаторам надо обращаться подчас через длинную цепочку названий стандартных классов-наследников.

Уровень такой иерархии в CLR называется пространством имен — в его текущих пределах и определяется некоторый ряд идентификаторов (например, типов данных). Эта концепция реализована и в языке Delphi для платформы .NET.

Проект Delphi задает так называемое *пространство имен* проекта по умолчанию. Все включенные в него модули вносят свой вклад в это пространство (предоставляют идентификаторы, описанные в их интерфейсных частях). Название этого пространства разработчик выбирает сам, указывая его следом за ключевым словом `Program`, например:

```
Program MyProjects.MyWork.Proj1;
```

Все входящие в такой проект модули будут по умолчанию считаться относящимися к пространству имен `MyProjects.MyWork.Proj1`, которое по отношению к ним должно использоваться как префикс. Так, модуль с заголовком:

```
unit Unit1;
```

в пределах данного проекта будет задавать пространство имен

```
MyProjects.MyWork.Proj1.Unit1.
```

Можно указать и явное пространство имен, формируемое неким модулем, если оно не должно быть привязано к текущему проекту:

```
Unit AnotherProjects.Lib.Additional.TestUnit;
```

Отличительная особенность пространства имен — запись его через точки, разделяющие отдельные идентификаторы. По правилам технологии .NET запись `AnotherProjects.Lib.Additional` определяла бы класс `AnotherProjects`, в котором имеется вложенное поле `Lib` (либо `Lib` как наследник класса `AnotherProjects`), которое, в свою очередь, детализируется понятием `Additional` и так далее. В Delphi запись названия пространства имен характерна тем, что точки-разделители являются неотъемлемой частью одного целостного названия, а идентификаторы между точками сами по себе не несут никакого смысла в плане поддержки реализации иерархии классов. Такая реализация уже существует в .NET, а схема записи пространств имен Delphi с дополнительными точками между идентификаторами создана прежде всего для удобства и совместимости с принятой формой записи .NET.

Так, вышеописанный модуль будет храниться в файле `AnotherProjects.Lib.Additional.TestUnit.pas` — такой подход просто повышает наглядность проекта. Конечно, не исключено, что в каких-то ситуациях будут существовать и файлы `AnotherProjects.Lib.Additional.pas`, и `AnotherProjects.Lib.pas`, реализующие функциональность или ее отдельные особенности вышестоящих уровней иерархии в рамках Delphi, однако это совсем не обязательно.

При использовании пространства имен требуется, чтобы перед идентификатором стояло либо полное наименование этого пространства, либо оно не использовалось бы вообще. Так, описание переменной `xyz` будет искаться

- компилятором как обычно — в пределах текущей процедуры или метода, текущего модуля, и в интерфейсных разделах всех модулей, перечисленных в разделе `uses`, а вот запись `AnotherProjects.Lib.Additional.TestUnit.XYZ` подразумевает обращение к идентификатору `XYZ` конкретного модуля (пространства имен) `AnotherProjects.Lib.Additional.TestUnit`. При этом будут неверны любые сокращенные формы записи:

```
TestUnit.XYZ
Lib.Additional.TestUnit.XYZ
```



Отметим, что в сборку .NET (точнее, в метainформацию о структуре сборки) включается информация об используемых в ней пространствах имен таким образом, что из названий пространств имен удаляется последний идентификатор с точкой перед ним. То есть, пространство имен модуля `AnotherProjects.Lib.Additional.TestUnit` будет представлено в метаданных соответствующей сборки проекта как `AnotherProjects.Lib.Additional` (без последнего названия модуля).

Одно и то же пространство имен может быть физически реализовано в виде нескольких файлов в формате .PAS. Для этого после названия пространства имен указывается ключевое слово `in`, за которым следует строка, а в ней через точку с запятой перечисляются модули, вся интерфейсная информация из которых включается в это пространство имен.

```
uses AnotherProjects.Lib.Additional in
'work/TestUnit.pas;..../add/GoUnit.pas';
```

Здесь пространство имен `AnotherProjects.Lib.Additional` задается интерфейсами модулей `TestUnit` и `GoUnit`. При этом если в них обнаружатся одинаковые идентификаторы, то компилятор сообщит о конфликте имен.

4.1.3. Дополнительные спецификаторы видимости

В целях синхронизации возможностей Delphi с требованиями спецификации общего языка CLS .NET в Delphi введены два дополнительных спецификатора видимости элементов класса. Комбинация `strict private` делает элемент видимым исключительно в рамках класса, в котором он описан. А элемент со спецификатором `strict protected` делает элемент класса видимым как в этом классе, так и во всех его наследниках.

```
type
  TWinForm = class (System.Windows.Forms.Form)
  ...
  strict private
    { Strict Private Declarations }
    x: integer;

end;
```

4.1.4. Абстрактные и закрытые классы

В дополнение к абстрактным методам (реализация которых в текущем классе не обязательна) в Delphi стало возможным описывать целиком абстрактные классы.

```
type Tx = class abstract

    public
        x,y: Integer;

end;
```

Экземпляры абстрактного класса создавать не разрешается, но допустимо определять на их основе классы-наследники.



Абстрактные классы активно применяются в технологии ECO, где они нужны для построения моделей на базе UML-диаграмм.

Класс также можно объявить закрытым с помощью ключевого слова `sealed`.

```
type Tx = class sealed

    public
        x,y: Integer;

end;
```

Экземпляры закрытого класса, наоборот, создавать разрешается, но нельзя определять классы-наследники.

4.1.5. Помощники классов

Помощники классов упрощают расширение описаний классов, представляя собой описание дополнительного «кусочка» уже существующего класса. Например, имеется описание:

```
type Tx = class

    public
        x,y: Integer;

end;
```

Мы хотим расширить его новым методом. Однако модифицировать описание этого класса по каким-то причинам нежелательно или невозможно. Но тем не менее расширить его можно использованием помощника класса. Он представляет собой описание, напоминающее обычное описание класса, с дополнительным указанием названия расширяемого класса после ключевых слов `class helper for`.

```
type TxHelper = class helper for Tx
```

```
    procedure CallMemo;
end;
```

Реализация метода `CallMemo` должна выполняться для помощника класса `TxHelper`:

```
procedure TxHelper.CallMemo;
begin
    ...
end;
```

Однако обращаться к нему можно как к обычному методу класса `Tx`:

```
var x: Tx;

x := Tx.Create;
x.y := 1;
x.CallMemo;
```

4.1.6. Поля класса

Поля класса — это его элементы, к которым может быть организован доступ независимо от наличия физической ссылки на реально существующий объект (экземпляр класса). Для этого внутри описания класса создается так называемый `var`-блок, определяющий подобные поля класса. Такой блок начинается комбинацией ключевых слов `class var`, и считается синтаксически завершенным, когда в дальнейшем встречается другой `var`-блок, описание процедуры или функции, свойства, конструктора, деструктора или любого спецификатора видимости.

```
type Tx = class

    class var
        x: integer;

end;
```

Здесь внутри типа `tx` мы описали поле класса под названием `x`. К нему в коде можно обращаться, например, так.

```
Tx.x := 1;
```

При этом идентификатор-префикс `tx` выступает здесь как название типа (класса), а не как реальный объект-экземпляр класса.

Схожий подход допустим и при создании статических методов класса, которые также могут быть вызваны, если экземпляр класса явно не создавался. Для этого в начале метода ставится ключевое слово `class`, а после описания метода указывается ключевое слово `static`.


```

type Tx = class

  class var
    x: integer;

  class function ab: Integer; static;

end;

// реализация:
class function Tx.ab: Integer;
begin
  Result := 1;
end;

//
Tx.x := Tx.ab;

```

При этом в реализации такого статического метода обращение к нестатическим элементам своего класса не допускается, так как доступа к реальному экземпляру класса (Self) у них нет.

4.1.7. Внутренние типы и константы классов

Внутри классов разрешается определять константы.

```

type Tx = class

  public
    const Len = 5;

end;

```

Размещать их имеет смысл, конечно, в разделе public, чтобы потом обращаться из внешних процедур.

```
Write( Tx.Len )
```

Также можно определять и внутренние типы.

```

type Tx = class

  public
    const Len = 5;

    type txRec = record
      x,y: Integer;
    end;

end;

```

...

```
var t: Tx.txRec;  
begin  
    t.y := 0;  
end;
```

В данном случае вложенный тип `txRec`, если он описан в разделе `public`, может использоваться в программе наравне с любыми другими типами, только с префиксом типа-владельца. Область действия таких типов ограничена только описанием класса, в котором они декларированы.

4.2. Новые возможности вызова подпрограмм

В ряде случаев в целях оптимизации можно попытаться объявить некоторые процедуры встраиваемыми (`inline`). Вызов таких процедур происходит фактически на этапе компиляции: вместо реализации вызовов в соответствующие точки кода автоматически подставляются исходные операторы встраиваемой (`inline`) подпрограммы. Впрочем, компилятору далеко не всегда удастся выполнить такую подстановку (например, если встраиваемый (`inline`) метод представляет собой конструктор, то необходимо учитывать логику работы приложения, что на этапе компиляции редко возможно), но в ряде достаточно очевидных случаев (например, если некоторая вычислительная функция со сложной логикой вызывается очень часто) такой подход может принести существенную выгоду.



Отметим, что выигрыш от встраиваемых подпрограмм выше при использовании компилятора .NET и при описании в качестве встраиваемых функций с большим количеством параметров. А маленькие функции, описанные как `inline`, могут общую производительность и понизить.

Чтобы объявить некоторую процедуру встраиваемой (`inline`), надо после ее определения поставить ключевое слово `inline`:

```
function Add2(x, y: Integer): Integer; inline;  
begin  
    Result := x + y;  
end;
```



С методами классов директива `inline` не работает.

Компилятору можно подсказать с помощью инструкции `{ INLINE AUTO }`, что весь последующий код он может обрабатывать, самостоятельно решая, какие подпрограммы использовать как встроенные. Также можно явно побудить

компилятор постараться сделать все без исключения подпрограммы встраиваемыми директивой {`INLINE ON`}. Отключается такой режим директивой {`INLINE OFF`}.

4.2.1. Начальные значения параметров функций

При описании заголовка процедуры или функции можно указывать начальные значения параметров в форме, принятой для определения констант.

```
function Add2(x: Integer = 1; y: Integer = 2): Integer;  
begin  
    Result := x + y;  
end;
```

Если теперь указать вызов такой функции без параметров:

```
Z := Add2();
```

то она вернет значение 3 (значение параметра `x` будет по умолчанию 1, а значение параметра `y` — 2).

Можно также указать не все параметры, при этом считается, что опускаются только последние из них.

```
Z := Add2(5);
```

Значение 5 будет передано параметру `x`, и функция вернет значение $5+2=7$.

Значения параметров по умолчанию необходимо указывать без пропусков: если некоторый параметр имеет значение по умолчанию, то и все последующие за ним параметры также должны иметь начальные значения. То есть, запрещены записи, подобные следующей:

```
function Add2(x: Integer = 1; y: Integer): Integer;
```

4.2.2. Динамические многомерные массивы

Удобная возможность Delphi по поддержке динамических одномерных массивов (векторов переменной длины) в новой версии расширилась средствами организации многомерных динамических массивов. Ранее динамический массив создавался так.

```
var a: array of Integer;
```

Теперь число его размерностей может задаваться не менее гибко, длиной конструкцией.

```
array of array of array of ... тип
```

Например, двумерный динамический массив может быть декларирован так.

```
var a: array of array of Integer;
```

Чтобы работать с таким массивом, сначала надо указать размер по первому измерению (фактически, количество одномерных массивов).

```
SetLength(a, 10);
```

А затем надо указать длину каждого из этих десяти одномерных массивов.

```
for i := 0 to 9 do  
    SetLength( a[i], 20 );
```

Вместо явных значений 0 и 9 (нижней и верхней границ массива) лучше использовать стандартные функции Low и High.

```
for i := Low(a) to High(a) do  
    SetLength( a[i], 20 );
```

Обращение к отдельным элементам массива происходит обычным способом.

```
for x := Low( a[3] ) to High( a[3] ) do  
    a[3,x] := 0;
```

Отметим, что размеры динамических массивов, в отличие от статических, по последнему измерению совсем не обязательно должны совпадать.

```
for i := 0 to 9 do  
    SetLength( a[i], i+2 );
```

Здесь мы формируем двумерный массив, в котором будет 10 строк, но длина каждой из строк (количество элементов одномерного массива соответствующей строки) составит от 2 до 11 элементов.

4.2.3. Перебор элементов контейнера

Очень удобна возможность элегантного перебора в цикле всех элементов некоторого контейнера-хранилища данных. В качестве контейнера может выступать массив, строка, множество или коллекция (с поддержкой метода GetEnumerator). Общая структура нового оператора цикла такова.

```
for элемент in контейнер do  
    тело-цикла;
```

В качестве элемента должна указываться переменная, тип которой совпадает с типом отдельного элемента контейнера. Значение этой переменной внутри тела цикла модифицироваться не может. Первоначально элемент цикла принимает значение первого элемента контейнера (расположенного по его нижнему индексу), а затем поочередно принимает значения следующих элементов до конца контейнера. Если контейнер — многомерный массив, то в элемент цикла будет записываться массив с числом измерений, на единицу меньшим, нежели исходный.

Если в качестве контейнера выступает строка, то перебор происходит поочередно по символам этой строки от начала до конца.

```
S := '12345';
```

```
For c in S do
```

```
  WriteLn( c ); // поочередно печатаем символы 1, 2, 3, 4, 5
```

Схожая ситуация и с множеством, а также с классами (и их наследниками), которые могут рассматриваться как последовательные наборы объектов. Для этого они должны быть наследниками стандартных классов `TCollection`, `TStrings`, `TInterfaceList`, `TComponent`, `TMenuItem`, `TCustomActionList`, `TFields`, `TListItems`, `TTreeNode`s или `TToolBar`. Например, перебор всех элементов стандартного компонента-списка `ListBox1`, размещенного на форме, и перенос их в многострочное поле `Memol` запишется так:

```
var s: String;
```

```
...
```

```
for s in ListBox1.Items do
```

```
  Memol.Lines.Add(s);
```

Так как свойство `Items` списка имеет тип `TStrings` (набор строк), то переменная `s` будет последовательно, по мере выполнения цикла, принимать значения из этого списка.

Переменная-счетчик в таких циклах не обязательно должна быть базового типа. Если коллекция хранит записи или объекты, то и счетчик будет описан как запись или класс.

4.2.4. Перегрузка операций

Очень мощная возможность перегрузки различных арифметических, логических и битовых операций, схожая с аналогичными возможностями `Си++`, открывает перед пользователями Delphi большие перспективы — гораздо более внушительные, нежели может показаться на первый взгляд. Так, сама по себе перегрузка операций позволяет с помощью стандартного синтаксиса записывать в общепринятом виде выражения над произвольными типами данных. Однако на самом деле перегрузка операторов — значительно более глубокая концепция.

Допустим, мы хотим создать в программе новый класс, но выполнять действия над ним желательно не только обращением к его полям и методам, но и с помощью обычных «арифметических» операций, записываемых символами «+», «-» и тому подобных. При этом смысл таких операций может, конечно, кардинально отличаться от исходного. Пусть наш класс `TVector` представляет собой вектор произвольной длины (фактически, динамический массив). Мы хотим создать в программе операции сложения, вычитания, умножения и деления таких векторов с помощью наглядной записи.

```

type TVector = class

    public
    FVector: array of Double;

    constructor Create(len: Integer);

end;

constructor TVector.Create(len: Integer);
var i: Integer;
begin
    inherited Create;
    SetLength(FVector, len); // длина массива

    // обнуляем:
    for i := 0 to len-1 do
        FVector[i] := 0;
    end;

```

При создании объекта-вектора в конструкторе надо указать количество его элементов.

Теперь в описание класса необходимо включить указание на перегружаемую операцию сложения, применимую для экземпляров данного класса. Для этого с помощью ключевых слов `class operator` и одного из названий перегружаемой операции надо добавить соответствующий заголовок. Дополним описание класса также функцией `len`, возвращающей текущее число элементов в нем.

```

type TVector = class

    public
    FVector: array of Double;

    class operator Add( a,b: TVector ): TVector;

    constructor Create(len: Integer);
    function len: Integer;

end;

// длина вектора
function TVector.len: Integer;
begin
    Result := Length(FVector);
end;

```

Название перегружаемой функции-оператора `Add` для операции сложения выбирается не произвольно, а в соответствии с документацией (раздел `Operator Overloads.NET`). Так, название `Add` соответствует перегружаемому сложению (реально же может выполняться любое действие, которое разработчик определит для оператора `+`).

Реализация функции оператора `Add` может быть такой.

```
class operator TVector.Add(a, b: TVector): TVector;
var
    i, n: Integer;
    v: TVector;
begin
    n := Math.min(a.len, b.len);
    v := TVector.Create(n);
    for i := 0 to n-1 do
        v.FVector[i] := a.FVector[i] + b.FVector[i];
    Result := v;
end;
```

Здесь создается новый вектор с длиной наименьшего из векторов-параметров, затем в него заносятся попарные суммы соответствующих элементов векторов-параметров.

Схожим образом реализуется и операция умножения (перегружаемая функция-оператор `Multiply`).

```
type TVector = class

    public
        FVector: array of Double;
        class operator Add( a,b: TVector): TVector;
        class operator Multiply( a,b: TVector): TVector;

        constructor Create(len: Integer);
        function len: Integer;
end;

class operator TVector.Multiply(a, b: TVector): TVector;
var
    i, n: Integer;
    v: TVector;
begin
    n := Math.min(a.len, b.len);
    v := TVector.Create(n);
    for i := 0 to n-1 do
        v.FVector[i] := a.FVector[i] * b.FVector[i];
    Result := v;
end;
```

Аналогично создаются операции вычитания (Subtract) и деления (Divide). Основное преимущество такого подхода в том, что теперь мы получили новый выразительный прием записи арифметических выражений над переменными-векторами.

```
var a, b, c, d: TVector;  
begin  
  a := TVector.Create(5);  
  b := TVector.Create(10);  
  c := TVector.Create(50);  
  d := TVector.Create(0);  
  
  // заносим значения в эти векторы  
  // ...  
  
  // вычисляем по формуле:  
  d := (a + b) * c - b / a;
```



Приоритеты операций остаются стандартными, то есть в первую очередь рассчитываются выражения в скобках, затем выполняется «умножение» и «деление», а потом — «сложение» и «вычитание».

Данная возможность позволяет реализовывать и гораздо более сложную логику, которая станет «раскручиваться» исполнительной средой автоматически. Например, не очень трудно написать систему аналитического дифференцирования, когда некоторая операция внутри себя разворачивается в свой результат-производную по известным правилам, и такой процесс будет развиваться автоматически до получения конечной формулы, так как логика всех операций задана заранее. А запустить ее можно, например, перегрузив стандартный метод ToString и просто обратившись к нему из формулы, подлежащей дифференцированию.

```
Write( (x*x + 2*x + 5).ToString );
```

Если для переменных x определен свой оригинальный тип элемента аналитического выражения, а в перегруженных операторах учтены взаимодействия как таких типов друг с другом, так и с числовыми константами, то аналитическое вычисление выполнится исполнимой средой поддержки перегруженных операторов автоматически (с привлечением механизма рекурсии).

При этом операции умножения и сложения должны выдавать в некотором внутреннем виде результат дифференцирования параметров.

4.2.5. Перегрузка преобразований типов

Помимо перегрузки операторов система Delphi допускает и перегрузку операций преобразования типов. Явное преобразование формируется функ-

цией-оператором `Explicit`, а неявное — функцией `Implicit`, результатом которых является перегружаемый тип данных, используемый в преобразовании. Так, например, преобразование нашего вектора в целое число может соответствовать определению его длины, а преобразование к дробному типу может происходить в виде вычисления суммы его элементов.

```

type TVector = class

  public
    FVector: array of Double;
    class operator Add( a,b: TVector ): TVector;
    class operator Multiply( a,b: TVector ): TVector;

    class operator Explicit( a: TVector ): Double;
    class operator Explicit( a: TVector ): Integer;

    constructor Create(len: Integer);
    function len: Integer;

end;

class operator TVector.Explicit( a: TVector ): Integer;
begin
  Result := a.len; // приводим к целому типу через
  возвращение длины вектора
end;

class operator TVector.Explicit( a: TVector ): Double;
var
  sum: Double;
  i: Integer;
begin
  sum := 0;
  for i := 0 to Integer(a)-1 do
    sum := sum + a.FVector[i];
  Result := sum;
end;

```

Здесь в качестве верхней границы цикла мы используем число элементов вектора, которое получаем преобразованием его к целому типу.



Возможность перегрузки операторов возможна как для классов, так и для записей (record). В записях также разрешены конструкторы с непустым списком параметров, объявление неvirtуальных методов и описание статических методов и процедур.

4.2.6. Типы данных .NET

Все типы данных, используемые в системе Delphi для платформы .NET, приведены в соответствие с требованиями технологии .NET. Основное изменение заключается в том, что теперь любой базовый тип Delphi (например, тип `Integer`) существует не сам по себе в виде чисто скалярной характеристики, а является наследником класса `System.Object` и к нему применимы все методы последнего и промежуточных наследуемых классов. В частности, как и любой класс, `System.Object` имеет конструктор и деструктор, поэтому формально работа с данными типа `Integer` может происходить как с экземплярами соответствующего класса (в частности, тип данных `Integer` из Delphi отображается на классе `System.Int32` среды .NET). Но для удобства разработчика и преемственности версий не требуется явно вызывать конструкторы для переменных традиционных массовых, базовых типов. Как и раньше, достаточно описать переменную:

```
var N : Integer;
```

после чего можно сразу ее использовать, например с помощью оператора присваивания.

```
N := 1;
```

Однако теперь в Delphi переменная `N` уже не считается обычной скалярной переменной, а представляет собой экземпляр класса `System.Int32` (хотя физически, на уровне реализации, компилятор работает с ней, скорее всего, по старинке). Поэтому мы можем задействовать все возможности соответствующего класса, в частности, удобный стандартный унаследованный метод `System.Object.ToString` по преобразованию содержимого любого класса в читаемый текст. Так, для получения текстового (строкового) представления значения `N` достаточно обратиться к его методу `ToString`:

```
N.ToString
```

Этот метод — функция, возвращающая строку-представление целочисленного значения `N`.

Соответствующие изменения затронули и `String`, старый строковый тип Delphi. Теперь он стал копией одноименного типа `System.String`.NET и дополнился рядом полезных методов. Так, метод-функция `Length` возвращает длину строки, а конкретный символ выделяется методом `Chars` (нумерация символов начинается с нуля). Можно также выделить подстроку (метод `CopyTo`), выполнить форматирование (`Format`), найти вхождение подстроки (`IndexOf`), вставить (`Insert`) или заменить (`Replace`) символы и так далее.

Более подробную информацию о новых возможностях базовых типов можно найти в документации по технологии .NET.

4.2.7. Динамически размещаемые массивы .NET

При создании приложений для технологии .NET можно использовать еще одну возможность динамического размещения массивов в памяти во время выполнения программы — в дополнение к описанным выше динамическим многомерным массивам. Такие динамически размещаемые массивы отличаются от многомерных динамических массивов тем, что их длина по всем измерениям всегда фиксирована, а от статических — тем, что размеры по каждому из измерений на этапе компиляции неизвестны и могут вычисляться непосредственно во время работы приложения. При описании такого массива конкретные значения числа элементов по каждому измерению не указываются, однако синтаксис с квадратными скобками и запятыми, разделяющими измерения, сохраняется.

```
var a: array [,] of Integer;
```

Здесь мы описали двумерный массив, который в процессе работы программы может быть сформирован (размещен в памяти) с помощью команды New.

```
a := New( array [2,3] of Integer );
```

В результате массив a станет матрицей размером 2×3 элемента. Причем размеры массива можно указывать в виде выражений.

```
N := 3;  
a := New( array [N, N+1] of Integer );
```

В функции New следом за описанием структуры массива можно задавать и список его начальных значений:

```
a := New( array [2,3] of Integer, ((1,2,3),(4,5,6)) );
```

При желании можно воспользоваться и процедурой SetLength с перечислением длин массива по каждому измерению:

```
SetLength( a, 2, 3 );
```

Последний вариант будет работоспособен при использовании компилятора для Win32. Как уже говорилось, возможность задействовать ключевое слово New при создании многомерных динамических массивов доступна лишь в .NET-версии.

4.2.8. Атрибуты классов

В каждой сборке .NET хранится метаинформация о самой этой сборке (пространства имен, структуры используемых классов и прочее). Delphi предоставляет разработчику возможность расширения этой метаинформации, формируемой компилятором (доступ к ней возможен и в процессе работы программы) собственными сведениями: пользовательскими атрибутами классов. Для этого надо объявить такой атрибут способом, схожим с декла-

рацией классов (наследуя свойства класса `TCustomAttribute`), после чего задействовать его и при необходимости обращаться к нему через стандартные .NET-интерфейсы.

Создадим пользовательский атрибут сборки `TMetaAttribute` по схеме обычного класса с одним свойством `X`.

```

type
    TMetaAttribute = class(TCustomAttribute)
        private
            FX : integer;
            procedure SetX(x2 : integer);
        public
            constructor Create(const myVal : integer);
            property X : integer read FX write SetX;
    end;

constructor TMetaAttribute.Create(const myVal: integer);
begin
    inherited Create;
    FX := myVal;
end;

procedure TMetaAttribute.SetX(x2: integer);
begin
    FX := x2
end;

```

Следующий шаг — инструкция компилятору о необходимости включения нового атрибута в сборку. Для этого перед описанием какого-либо класса, который мы хотим снабдить нашим атрибутом, надо указать название атрибута в квадратных скобках, а для самого атрибута — задать значения его свойств, начиная с параметров конструктора. То есть, если мы создаем класс `Tx`:

```

type Tx = class
    ...
end;

```

и хотим снабдить его дополнительным атрибутом, то перед началом описания `Tx` надо указать:

```

[TMetaAttribute(567, X = 12)]

type Tx = class
    ...
end;

```

Отметим, что описание атрибута и связанного с ним класса может быть размещено в интерфейсной части модуля.

Теперь из кода программы можно обращаться к списку атрибутов экземпляра класса Tx, и в нем будет присутствовать пользовательский атрибут.

```
var
  A: Tx;
  TS: System.Type;
  AList: array of TObject;

begin
  A := Tx.Create;
  TS := A.GetType;
  Write( TS.FullName ); // FullName - стандартный метод
  получения полного названия типа
  AList := TS.GetCustomAttributes(True); // получили список
  атрибутов
```

Далее можно обращаться к любому из элементов списка AList, а тип любого из элементов списка узнается с помощью функции GetType().

4.2.9. Управляемый и неуправляемый код

Код, отвечающий требованиям общезыковой среды CLR выполнения .NET-программ и способный обращаться к ее службам, называется *управляемым*. Практически все профессионально сделанные .NET-приложения базируются на управляемом коде, хотя не всегда такой код будет гарантированно безопасным. При создании .NET-приложений компилятор Delphi по умолчанию генерирует управляемый код. Отметим, что в поставку Borland Development Studio 2006 также входят .NET-компиляторы C# и VB.NET. При этом имеется возможность загрузки и переноса в Delphi проектов Microsoft Visual Studio, написанных на этих двух языках. В пакет BDS 2006 входит также среда программирования на языке C++ (C++Builder), но трансляция программ, написанных на этом языке, в управляемый код сопряжена со значительными сложностями, а в ряде ситуаций невозможна. Так, например, в технологии .NET не допускается множественное наследование и введены существенные ограничения на работу с указателями. Поэтому компилятор C++Builder можно использовать лишь для создания программ Win32.

В то же время, управляемая программа для платформы .NET может вызывать и неуправляемые функции, например из библиотек DLL, подготовленных для платформы Win32. Для этого в Delphi реализован так называемый *интерфейс виртуальной библиотеки (Virtual Library Interface)*. Он базируется на сервисе .NET, который называется *Platform Invoke (вызов платформно-зависимого кода)*. Этот сервис должен быть реализован в исходных текстах прямой поддержкой — описанием вызываемых функций, которое обрабатывается во время компиляции (так называемая *технология DLLImport*). Для этого перед заголовком функции надо указать атрибут DLLImport, а в его параметрах — название файла библиотеки DLL и название точки входа в библиотеку, что выполняется настроечным параметром EntryPoint.

```
1/ [DllImport('win32lib.dll', EntryPoint = 'EntryCall32')]  
   function win32call(x: Integer): Integer; external;
```

Здесь функция `win32call` из библиотеки `win32lib.dll` с неуправляемым кодом, вызываемым через точку входа в библиотеку `EntryCall32`, становится доступной в программе для .NET.



Чтобы атрибут `DLLImport` стал доступен для использования в исходном коде, надо в список подключаемых модулей раздела добавить обращение к пространству имен `System.Runtime.InteropServices`.

Однако интерфейс `Virtual Library Interface` превосходит по своему потенциалу концепцию `DLLImport`. С его помощью можно обходиться без атрибута `DLLImport` и, более того, загрузка и вызов функций из библиотек DLL становятся возможны динамически на этапе выполнения программы. Этот подход полезен, когда размер подключаемой библиотеки велик и желательно ограничиться ее динамической обработкой. Его важное отличие от подхода `DLLImport` заключается в ненужности задания имени DLL-файла в исходном тексте.

Для работы с этим интерфейсом надо добавить в модуль пространство имен `Borland.Vcl.Win32`, определить пользовательский интерфейс к вызываемым функциям и вызвать функцию `Supports` этого пространства имен, чтобы проверить, возможно ли обратиться к прикладным функциям, описанным программистом в прикладном интерфейсе.

Для вышеприведенного примера новый вариант кода может быть таким.

```
uses Borland.Vcl.Win32;  
  
type IWin32Lib = Interface  
    function win32call(x: Integer): Integer;  
end;
```

Здесь мы описали интерфейс, через который во время работы программы будем обращаться к нужной библиотеке. Для этого надо воспользоваться упомянутой функцией `Supports`, в качестве параметров которой указываются название файла библиотеки DLL, название интерфейса и название переменной, реализующей этот интерфейс.

```
var  
    MyInt: IWin32Lib;  
...  
  
if Supports('win32lib.dll', IWin32Lib, MyInt) then  
  
    Write( MyInt.win32call(12) );
```

Отметим, что в среде Delphi для платформы .NET сохраняется возможность вставлять в код прямые вызовы функций API Win32 (только в спи-

ске стандартных подключаемых библиотек надо указать модуль Windows). И хотя их реализация осуществляется не совсем тривиально: через технологию маршallingа в обход вызовов, специфичных для СОМ-интерфейсов, — тем не менее с точки зрения прикладного разработчика никаких трудностей организация таких вызовов не создает.

4.3. Технологии отладки: новые возможности

Среда автоматически запускает отладчик Win32 или .NET в зависимости от типа создаваемого проекта. При этом везде, где возможно, сохраняется одинаковый интерфейс пользователя.

4.3.1. Точки прерывания

Окно списка точек прерывания, вызываемое командой View > Debug Windows > Breakpoints (Вид > Окна отладчика > Точки прерывания), дополнилось флажком временного отключения этих точек без необходимости их полноценного удаления. Кроме того, все точки теперь можно включать и отключать соответствующими кнопками на панели (рис. 4.1).

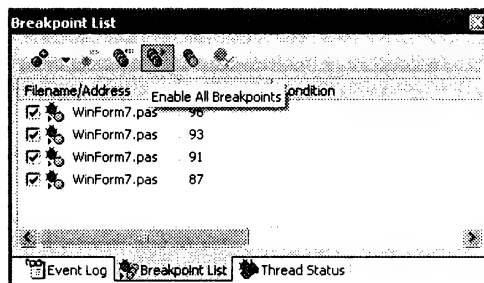


Рис. 4.1. Управление точками прерывания

Если ведется протокол отладки, то в него можно дополнительно включать информацию о стеке приложения в каждой из точек прерывания. Для этого в настройке соответствующей точки (настройка доступна командой Properties из контекстного меню этой точки) надо установить флажок Log Call Stack (Протоколировать стек вызовов) и выбрать значение Включить весь стек (Entire Stack) или Включить часть стека (Partial Stack) и дополнительно указать количество элементов стека в поле Number of frames (рис. 4.2).

4.3.2. Исключительные ситуации

Система Delphi позволяет настраивать способ реакции приложения на возникновение в нем той или иной исключительной ситуации или системного прерывания. Список обрабатываемых прерываний представлен в настройках отладчика Language Exceptions (Языковые исключения), доступных через

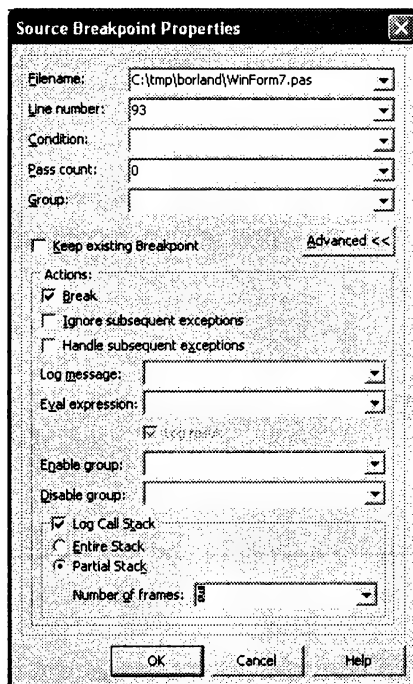


Рис. 4.2. Настройка данных, выдаваемых в точке прерывания

окно Tools ➤ Options ➤ Debugger Options ➤ Borland Debuggers (Сервис ➤ Параметры ➤ Параметры отладчика ➤ Отладчик Borland) и Native OS Exceptions (Собственные исключительные ситуации операционной системы). Прикладные события можно игнорировать, просто снимая флажок соответствующей группы, а для системных событий задается форма реакции (Handled by): перехват отладчиком (Debugger) или программой (User program), — и дальнейшее действие (On Resume): продолжение работы после обработки исключения (Run handled) или продолжение работы без обработки исключения (Run unhandled) — рис. 4.3.

4.3.3. Смешанный код

В окне машинного кода View ➤ Debug Windows ➤ CPU (Вид ➤ Окна отладки ➤ Процессор) теперь можно визуализировать любой смешанный код: как исходные операторы Delphi, так и машинный код конкретного процессора, а также команды виртуального ассемблера *MSIL*, в код которого транслируются программы .NET. Для этого надо включить флажки Mixed IL Code (Смешанный внутренний код) и Mixed Source Code (Смешанный исходный код) контекстного меню панели CPU (Процессор) — рис. 4.4.



Во всех окнах процесса отладки можно пользоваться кодировкой Unicode.

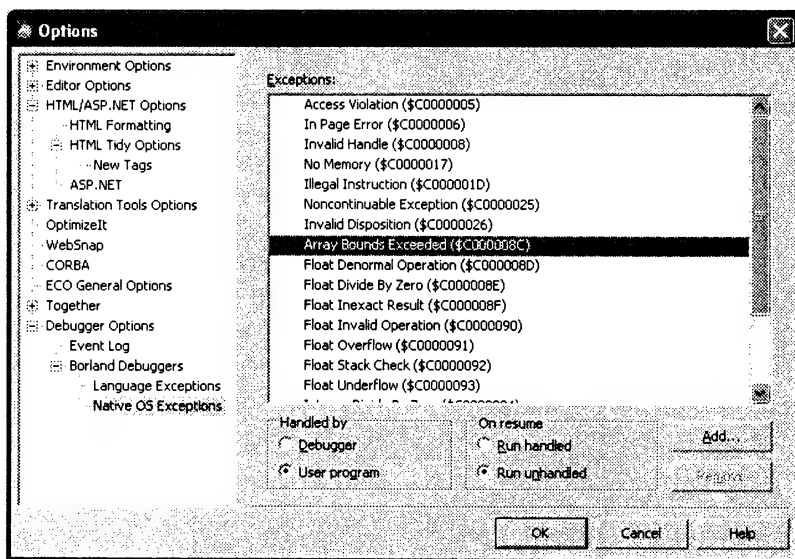


Рис. 4.3. Настройка списка обработки прерываний

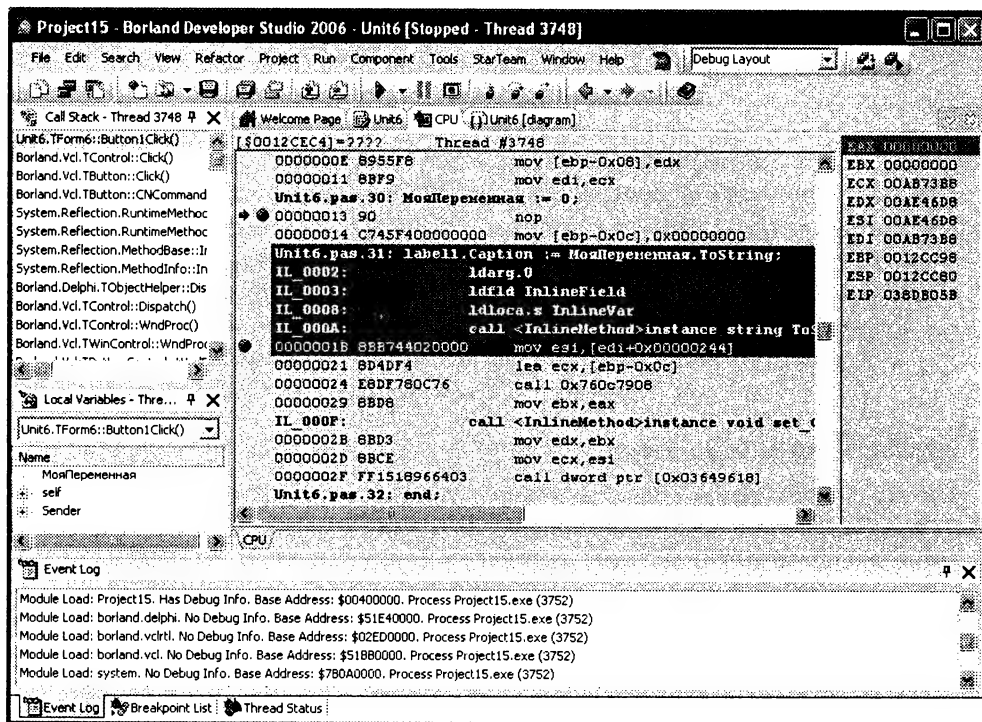


Рис. 4.4. Просмотр ассемблерного кода

4.3.4. Загрузка символических таблиц

Система Delphi автоматически формирует и загружает все символические таблицы, необходимые для отладки текущего проекта, выполняя этот процесс незаметно для разработчика. В Delphi 2006 появились средства управления этим процессом.



Символическая таблица нужна для поддержки процесса символьной отладки на уровне исходного кода. В ней хранятся взаимосвязи между символическими идентификаторами, используемыми в программе (например, именами переменных), и их адресами в оперативной памяти при работе приложения.

По умолчанию, Delphi загружает все обнаруженные символические таблицы. В новой версии можно вручную ограничить перечень этих таблиц, что в случае крупных проектов позволяет существенно ускорить процесс отладки. Компилятор Delphi для Win32 хранит все символические таблицы в оперативной памяти, и лишь в случае дистанционной отладки с другого компьютера может формировать внешние файлы с расширением .RSM. Компиляторы для платформы .NET (языки Delphi, C# и VB.NET) создают файлы PDB, а компилятор Си++ — файлы .TDS.

Параметры процесса поиска символических таблиц задаются в настройках проекта Project > Options > Debugger Options > Symbol Tables (Проект > Параметры > Отладчик > Символические таблицы). В поле Debug symbols search path (Путь доступа к таблицам) можно задать набор каталогов и указать последовательность их просмотра. А если требуется загружать не все таблицы, то надо снять флажок с поля Load all symbols (Загружать все символы) и вручную выбрать модули, подлежащие анализу. Кроме того, глобальные каталоги поиска указываются и в настройках среды Tools > Options > Debugger Options > Borland Debuggers (Сервис > Параметры > Параметры отладчика > Отладчик Borland), в поле Debug symbols search path (Путь доступа к таблицам).

4.3.5. Разное

Окно модулей приложения, вызываемое командой View > Debug Windows > Modules (Вид > Окна отладчика > Модули), представляет все процессы, контролируемые отладчиком, и модули, загруженные для каждого процесса. По ним можно просмотреть и упорядочить иерархическое представление классов, методов и пространств имен (рис. 4.5).

Окно локальных переменных, доступное по команде View > Debug Windows > Local Variables (Вид > Окна отладчика > Локальные переменные) теперь позволяет просматривать значения не только переменных текущего метода и модуля, но и переменных из любых других блоков приложения (они выбираются в раскрывающемся списке в верхней части окна) вне зависимости от платформы. Ранее это было возможно лишь при отладке приложений .NET. Кроме

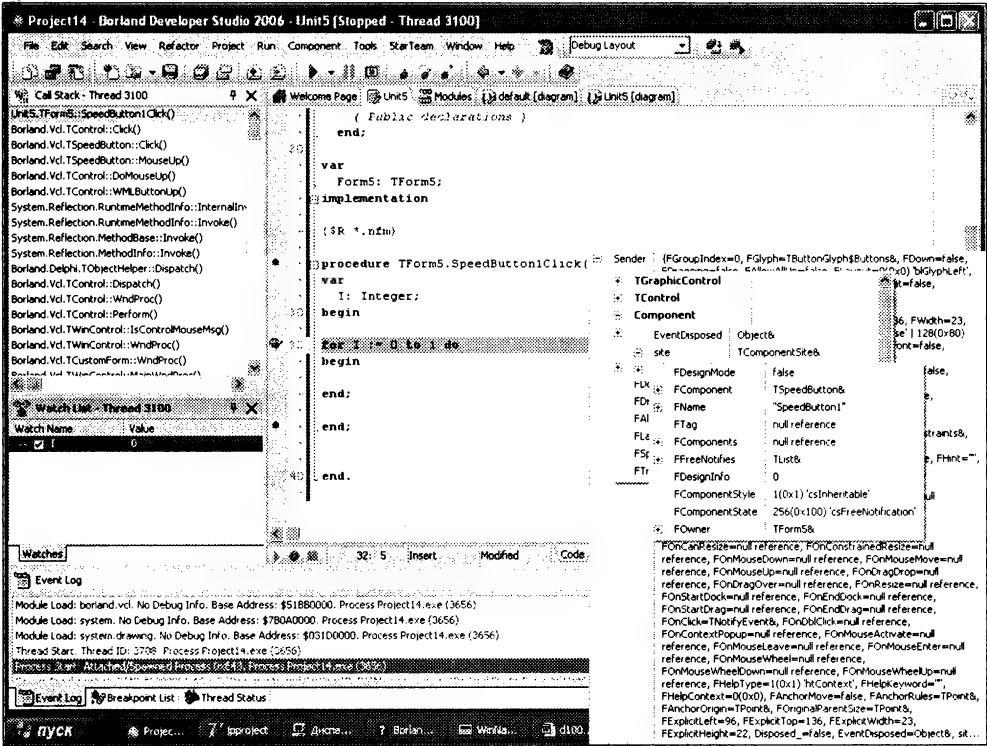


Рис. 4.7. Просмотр свойств отдельных объектов

Подобный сервис раскрытия объекта реализован даже для всплывающих подсказок, возникающих в процессе пошаговой отладки, при наведении курсора на идентификатор.

Глава 5. Технологии работы с базами данных

5.1. Технология работы с базами данных BDP.NET

5.1.1. Визуальный Проводник данных (Data Explorer)

Создание программы, взаимодействующей с базами данных, начинается, как правило, с подготовки нового соединения с выбранной СУБД. В версии Delphi для среды .NET вместо устаревшей и автономной утилиты SQL Explorer, ориентированной на устаревшую технологию BDE, реализован встроенный Проводник данных (Data Explorer) со значительно расширенными возможностями и в то же время упрощенным управлением.

Проводник данных обычно находится на вкладке Data Explorer — основной панели Менеджера проектов — или может быть активизирован командой View > Data Explorer (Вид > Проводник данных). Он предоставляет доступ к настройкам соединений как для технологии BDP.NET, так и для технологии dbExpress, которая может быть задействована в приложениях VCL и VCL.NET.

Для создания нового соединения выберем поставщика данных, допустим, СУБД Interbase, в контекстном меню выберем пункт Add New Connection (Добавить новое соединение), после чего укажем нужного поставщика, введем название соединения и выполним остальные требуемые настройки. При желании новое соединение можно раскрыть и просмотреть внутреннюю структуру базы, изучить построение той или иной таблицы (рис. 5.1). Можно даже увидеть хранящиеся в таблице данные, если дважды щелкнуть на ее названии: в основном рабочем окне появится содержимое записи, доступное для редактирования (рис. 5.2).

Проводник данных обладает интересной возможностью формирования соединения с базой данных в визуальном режиме. Сделать это проще всего в полуавтоматическом режиме: сохраняя нужное соединение данных открытым в Проводнике, выберем в нем любую доступную таблицу и просто перетащим ее на форму текущего открытого приложения .NET. В результате в проект добавятся элементы BdpConnection1 и BdpDataAdapter1, а их параметры соединения с базой данных будут настроены автоматически.

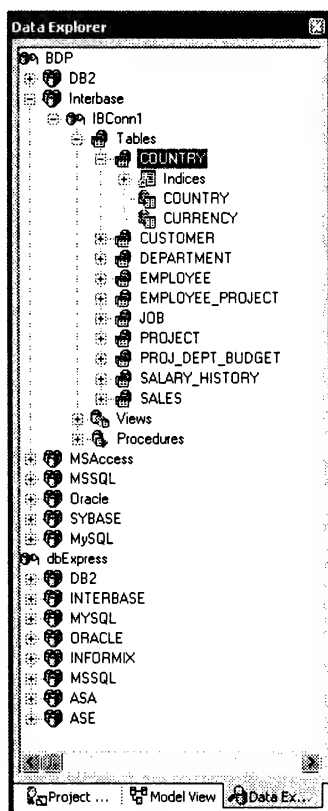


Рис. 5.1. Представление структуры данных в Проводнике

Окно Проводника данных дополнилось новыми командами контекстного меню для таблиц баз данных: Copy/Paste Table (Копировать/вставить таблицу) и Migrate Data (Перенос данных). Кроме того, в контекстном меню появились удобные команды работы с метаданными: можно создавать новую таблицу (New table), загружать данные из таблицы (Retreive Data From Table) или запроса (Retreive Data From View), удалять таблицы (Drop Table), модифицировать структуру таблиц (Alter Table), настраивать хранимые процедуры и запускать их тестовое выполнение.

5.1.2. Технология Borland Data Providers for .NET (BDP.NET)

В состав среды .NET Framework входит стандартная технология универсального доступа к данным ADO.NET, ориентированная на работу с СУБД Microsoft SQL Server, Oracle и другими, а также с компонентами OLE DB. С помощью технологии ADO.NET в среде .NET обрабатываются данные любых прикладных приложений — от Windows Forms до Web-служб.

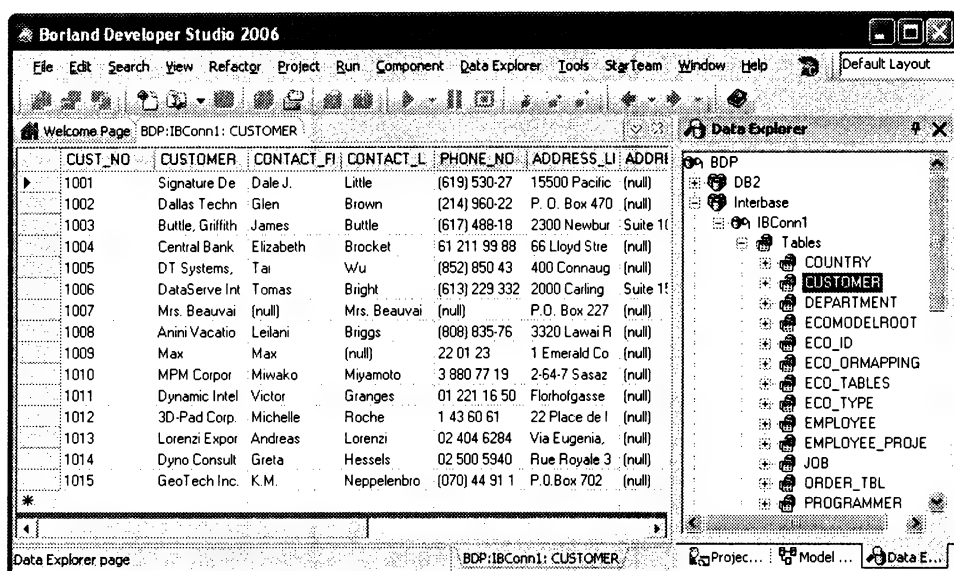


Рис. 5.2. Доступ к содержимому таблиц баз данных из Delphi

Корпорация Borland включила в Delphi собственные .NET-средства (провайдеры) доступа к базам данных, которые называются *Borland Data Providers for .NET (BDP.NET)*. Они ускоряют процесс программирования и позволяют работать как с СУБД MS SQL Server и Oracle, так и с СУБД IBM DB2, Sybase, Borland Interbase и Microsoft Access. Эти провайдеры способны взаимодействовать с СУБД напрямую, в обход шлюза COM Interoperability (он обеспечивает обращение к COM-объектам из среды .NET), характерного для подходов OLE DB. В то же время, поставщики BDP.NET поддерживают более универсальную и более открытую модель доступа к произвольным реляционным платформам, нежели стандартные средства оболочки .NET Framework. Так, в дополнение к принятым в технологии .NET низкоуровневым типам элементов таблиц предлагаются собственные (принятые в SQL) логические типы данных, которые при необходимости автоматически транслируются в комбинации типов .NET и обратно, что делает возможным настройку SQL-запросов на конкретные модели СУБД. Кроме того, удобные сопроводительные средства провайдеров BDP.NET дают разработчику доступ к метаданным, упрощают создание схем передачи информации и обеспечивают другие потребности.



К обширному списку СУБД, поддерживаемых в Delphi 2006, добавились Interbase 7.5, Oracle 10g, IBM DB2 8.x, Microsoft SQL Server 2000/2005, Informix 9.x, SQL Anywhere 9, MySQL 4.0.24 и Sybase 12.5. Специально для последней модели СУБД создано пространство имен Borland.Data.Sybase.

В целях совместимости разработчик может использовать при создании полноценного настольного приложения (Windows Forms) для платформы .NET линейку компонентов dbExpress, но если создается новое приложение для работы с базами данных, то лучше все же воспользоваться новыми технологиями.

Основной элемент программы, предназначенной для работы с базами данных, — это набор данных (DataSet). Он представляет собой набор таблиц, каждая из которых в свою очередь состоит из строк и столбцов. Формируется набор путем передачи поставщику (некоторому промежуточному компоненту) SQL-запроса, который тот отправляет СУБД, получает от нее ответ и возвращает его обратно в форме DataSet.

Набор данных хранит в адресном пространстве прикладного (клиентского) приложения таблицы, полученные от СУБД, и предоставляет разработчику средства их гибкой обработки (в дополнение к стандартным средствам просмотра, сортировки и выполнения запросов). В частности, в одном наборе данных могут храниться копии таблиц и результаты работы запросов к совершенно разным физическим источникам данных. А поставщик данных следит за согласованностью информации в прикладной программе и на сервере.

В прикладной программе в качестве набора данных разработчик может использовать либо стандартный компонент DataSet платформы .NET, либо новый, собственный, типизированный набор данных (наследник класса DataSet), в котором структура задана конкретной раскладкой информации в определенной таблице базы. Последний подход более корректен, так как позволяет компилятору проверить типы данных каждого столбца таблицы и контролировать ход операций над ними. В таком случае вся подробная информация о таблице будет автоматически сформирована и записана в XML-файл проекта с расширением .XSD. При необходимости этот файл можно редактировать и вручную, не модифицируя код самой программы. Основным недостатком такого подхода — сложность перехода к поддержке дополнительных СУБД в программе, так как конкретные внутренние типы данных таблиц разных систем могут сильно различаться и их согласование способно привести к ошибкам преобразования типов.

5.1.3. Пример создания приложения BDP.NET

1. Создадим пустое настольное приложение (Windows Forms).
2. В проекте разместим компонент BdpConnection — основной элемент проекта, отвечающий за действующую связь с СУБД. Он напоминает компонент TSQLConnection из набора dbExpress. Этот компонент, как и другие, входящие в набор BDP.NET, расположен внутри группы Borland Data Provider палитры инструментов (рис. 5.3).
3. Выделим размещенный компонент BdpConnection в Инспекторе объектов и откроем Редактор соединений (Connection Editor).

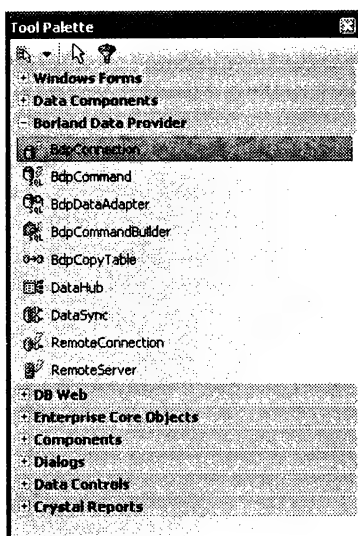


Рис. 5.3. Список компонентов BDP.NET

4. Выберем нужное соединение (пусть это будет соединение IBConn1, устанавливающее связь с СУБД Interbase, которая должна быть запущена, а соединение — настроено). При необходимости можно добавить новое соединение, нажав на кнопку Add (Добавить). Проверить работоспособность соединения можно с помощью кнопки Test (Тест) — рис. 5.4.

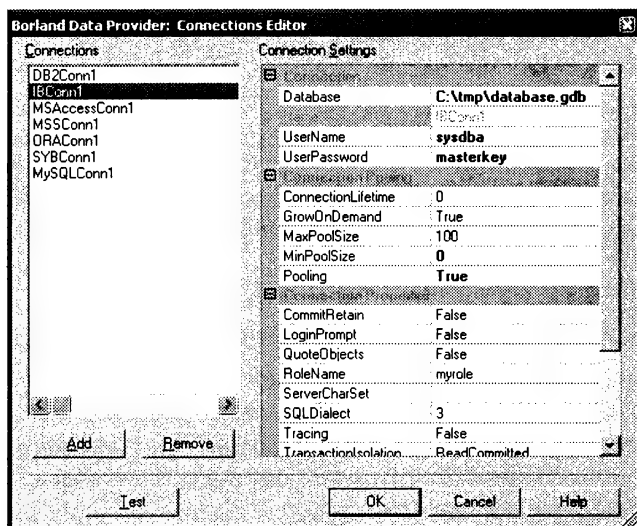


Рис. 5.4. Редактор соединений в действии

В Редакторе соединений появилась новая важная возможность управления пулом BDP-соединений (раздел *Connection Pooling*). С помощью представленных здесь настроек обычно удастся существенно повысить скорость соединения Delphi-программ с СУБД. Специальный Менеджер BDP-пула (*BDP Pool Manager*), автоматически встраиваемый в приложение, формирует отдельный пул для каждой оригинальной строки, описывающей соединение с СУБД (свойство *ConnectionString* компонента *BdpConnection*). Список доступных параметров настройки раздела приведен в таблице 5.1.

Таблица 5.1. Параметры настройки пула соединений с СУБД

Параметры настройки пула	Значение
Pooling	Если значение равно True, то в приложении будет задействован Менеджер BDP-пула
MinPoolSize	Минимальное количество соединений, которые будут храниться в пуле
MaxPoolSize	Максимально допустимое для пула количество соединений
GrowOnDemand	Если установлено значение True, то превышение максимального числа обслуживаемых соединений в пуле приведет к автоматическому формированию нового соединения, которое в этом пуле обслуживаться не будет. В противном случае при превышении возникает исключительная ситуация, которую надо обрабатывать программно
ConnectionLifetime	Максимальная продолжительность времени жизни соединения, в секундах. Если значение равно 0 (по умолчанию), то время существования соединения не контролируется

- На следующем шаге требуется настроить функционирование поставщика данных (адаптера), ответственного за соединение с конкретной СУБД. Для этого предназначен компонент *BdpDataAdapter*. Разместите его, выделите и обратитесь к Инспектору объектов. В разделе *Fill* для свойства *SelectCommand* (команда, которую поставщик будет передавать СУБД для формирования набора данных) следует выбрать подсвойство *Connection*, связав через раскрывающийся список поставщиков данных с настроенным соединением (*BdpConnection1*).

Затем из контекстного меню компонента-адаптера вызовем редактор-конфигуратор SQL-выражений (команда меню *Configure Data Adapter*). Он немного напоминает SQL-редактор, ранее использовавшийся при работе с компонентами *dbExpress*. С помощью этого конфигуратора готовится SQL-код соответствующей команды формирования набора данных (рис. 5.5). Делается это следующим образом.

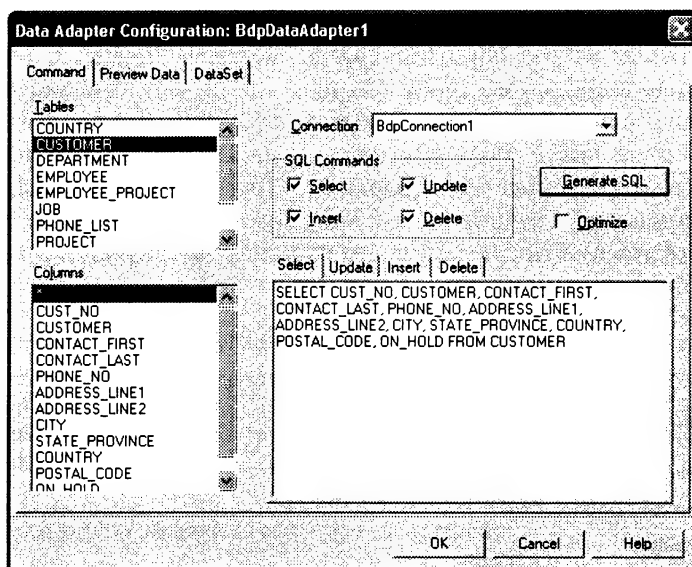


Рис. 5.5. Средство автоматизации подготовки SQL-кода

На вкладке **Command** выберем таблицу (список **Tables**), доступную в СУБД, с которой установлено соединение. В списке **Columns** выделим несколько столбцов, которые хотим получить в приложении в виде набора данных, и после нажатия на кнопку **Generate SQL** (Генерировать SQL-запрос) на вкладке **Select** появится SQL-код соответствующего запроса для получения нужных элементов таблицы, например, такой.

```
SELECT CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST,
PHONE_NO, ADDRESS_LINE1, ADDRESS_LINE2, CITY, STATE_PROVINCE,
COUNTRY, POSTAL_CODE, ON_HOLD FROM CUSTOMER
```

При желании результат этого запроса (набор данных, который поступит в нашу программу от СУБД) можно предварительно просмотреть на вкладке **Preview Data** (надо дополнительно нажать кнопку обновления **Refresh**) — рис. 5.6).

6. К данному моменту мы уже почти полностью сформировали работоспособный каркас приложения. Теперь данные, доступные с сервера, уже уверенно поступают в Delphi. Осталось явно передать их в нашу программу и каким-то образом сохранить в ней в виде локального набора. С этой целью перейдем в редакторе на вкладку **DataSet**. Если бы в нашем приложении уже существовал какой-либо объект, ответственный за обработку набора данных, он появился бы в данном окне (в раскрывающемся списке **Existing DataSet**). Но так как мы его пока не создали, нажмем кнопку **New DataSet** и введем названия объекта, например, **dataSet1** — рис. 5.7.

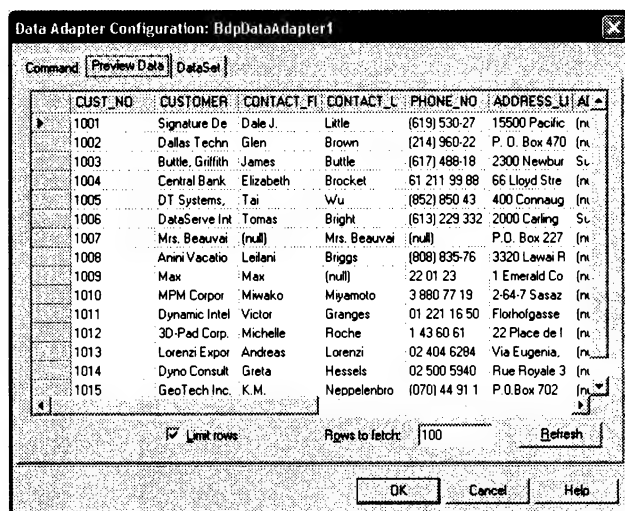


Рис. 5.6. Предварительный просмотр данных, поступающих от СУБД

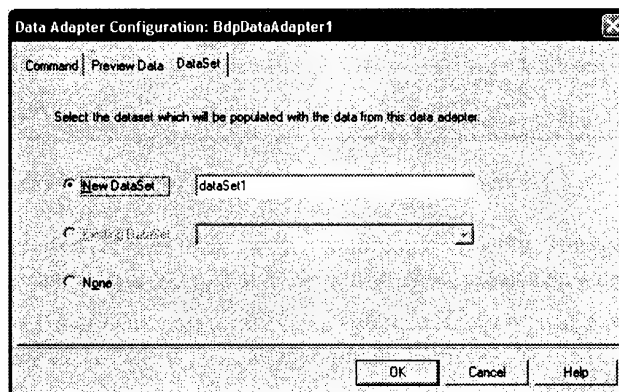


Рис. 5.7. Создание локального набора данных в приложении

Нажав теперь на кнопку OK, мы получим в проекте готовый компонент `dataSet1`, настроенный на наше соединение. Чтобы программа взаимодействовала с ним «вживую», зададим свойству `Active` объекта `BdpDataAdapter1` значение `true`. Теперь достаточно добавить на форму компонент визуализации набора данных (например, `DataGrid`), указав в его свойстве-источнике визуализируемых данных `DataSource` автоматически созданный компонент `dataSet1` — рис. 5.8.

Вариант источника данных `DataGrid` для платформы .NET допускает поставку сразу нескольких наборов, поэтому в свойстве `DataMember` этого компонента надо выбрать конкретный набор (например, `CUSTOMER`) — на форме

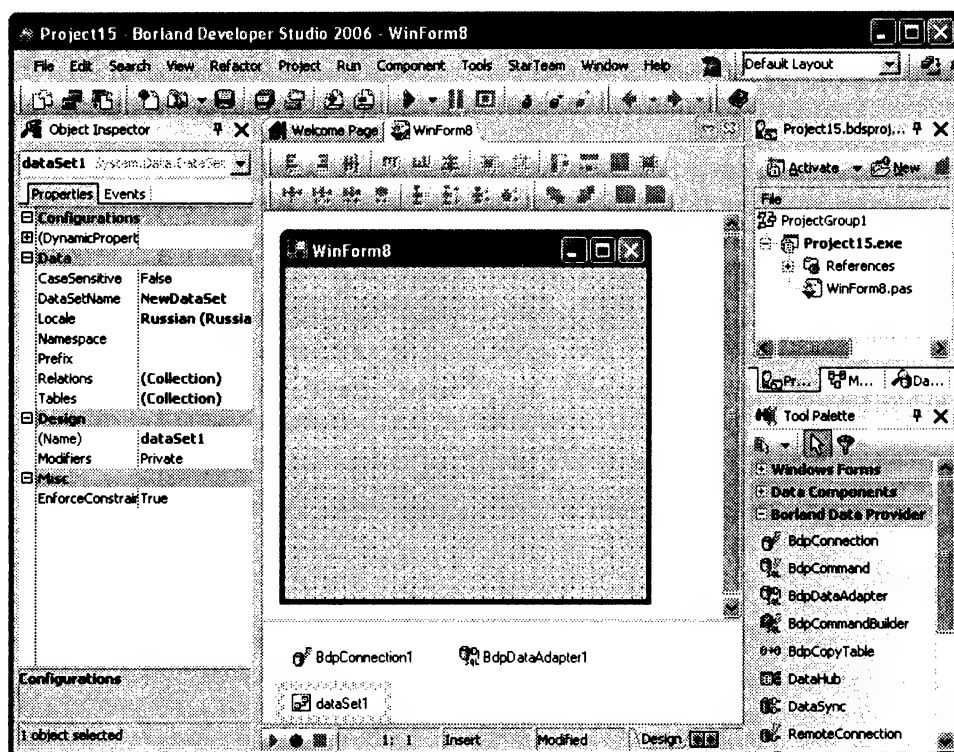


Рис. 5.8. Подготовка к визуализации полученного набора данных

сразу появится его содержимое, после чего можно запускать программу и проверять ее взаимодействие с СУБД в реальном времени (рис. 5.9).

Если же явно визуализируемую таблицу не указать, то в работающей программе будет отображено дерево всех доступных таблиц в рамках текущего соединения.

Таким образом, без какого-либо ручного кодирования мы создали полнофункциональное приложение, работающее с базами данных.

5.1.4. Способы вызова хранимых процедур

Когда в компоненте `BdpCommand`, реализующем возможность выполнения команд СУБД, в качестве типа команды (свойство `CommandType`) выбрано значение «хранимая процедура» (`StoredProcedure`), то в раскрываемом списке свойства `CommandText` можно выбрать имя любой из доступных процедур, хранимых в базе; их перечень сформируется автоматически. А в контекстном меню компонента `BdpCommand` с помощью пункта `CommandText Editor` задаются подробные настройки выполнения выбранной процедуры, имеется даже возможность ее тестового запуска кнопкой `Execute` (рис. 5.10).

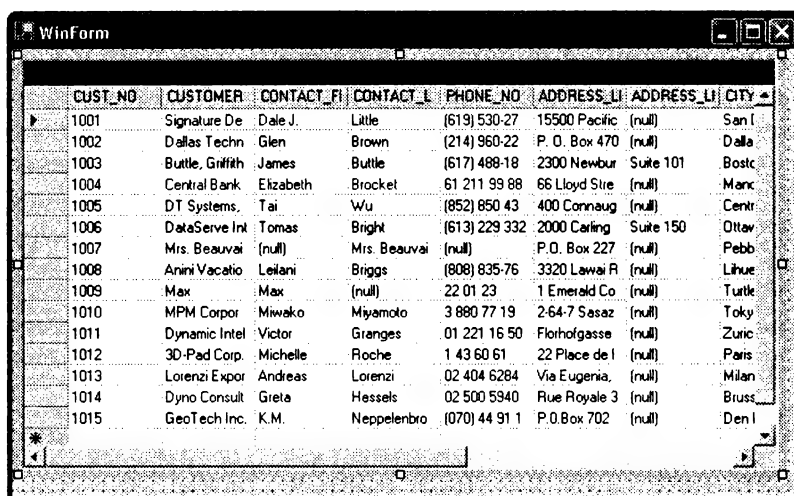


Рис. 5.9. Доступ к таблице базы данных на этапе проектирования

С помощью Проводника данных хранимые процедуры (раздел Procedures конкретной базы Проводника) можно перетаскивать в Дизайнер, в результате чего в проекте автоматически формируется связь с базой через компонент BdpConnection и настраивается компонент выполнения хранимой процедуры BdpCommand.

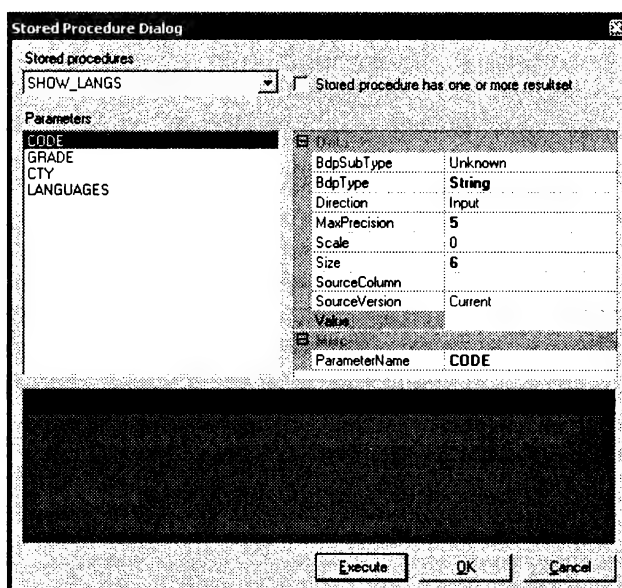


Рис. 5.10. Средство настройки хранимых процедур

5.1.5. Другие возможности

Компонент `VdpCopyTable` позволяет быстро запрограммировать процесс переноса содержимого таблиц из одной базы данных в другую. Это осуществляется заданием источника набора данных `SourceCommand` и принимающей таблицы `DestinationTable`. Компонент обеспечивает перенос схемы таблицы, первичных ключей и записанных в нее данных.

Компоненты `DataHub` и `DataSync` группы `Borland Data Provider` позволяют состыковать работу множества поставщиков данных в одном приложении и синхронизировать одновременную обработку множества таблиц. А добавив к этой паре компоненты `RemoteConnection` и `RemoteServer`, можно приступить к созданию многоуровневых приложений, в которых элементы серверной логики выполняются на разных компьютерах в сети.

Редактирование перечня и настроек таблиц, доступных в рамках некоторого набора данных `DataSet`, а также, что важно, взаимосвязей между таблицами, можно теперь выполнять в специальных редакторах, доступных через свойства `Tables` и `Relations` набора данных. В первом случае вызывается редактор перечня таблиц, а во втором появляется возможность визуально настраивать связи между таблицами.

5.2. Пример работы с базами данных dbGo (ADO.NET)

Набор компонентов `dbGo` объединяет в себе стандартную технологию `Microsoft ADO`. В целях обеспечения совместимости имеются версии этих компонентов как для приложений `Win32`, так и для платформы `.NET`. Они доступны при создании приложений `VCL` и `VCL.NET`.

1. Создадим пустое приложение `VCL Forms`. Разместим на форме компонент `TADOConnection` из группы `dbGo` палитры инструментов. Он выполняет ту же роль, что и компонент `VdpConnection` — связывает приложение с СУБД (рис. 5.11).

Настройка этого компонента выполняется в обычном для классических технологий `Windows` не слишком наглядном стиле. Двойным щелчком на нем открываем диалоговое окно формирования строки, описывающей соединение данного компонента с базой данных (рис. 5.12).

2. Сохраняя переключатель в положении `Use Connection String` (Использовать строку с описанием соединения), — нажимаем кнопку `Build` (Построить строку), после чего

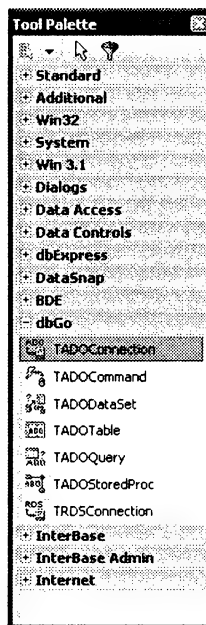


Рис. 5.11. Список компонентов `dbGo`

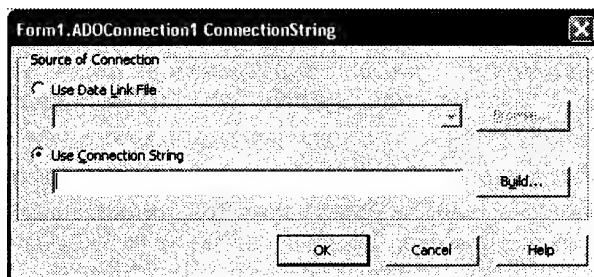


Рис. 5.12. Здесь формируется строка, описывающая соединение с базой данных

открывается окно со списком всех зарегистрированных в системе поставщиков OLE DB. Выберем, например, строку Microsoft Jet 4.0 OLE DB Provider — (рис. 5.13).

3. Нажимаем кнопку Далее и в следующем окне выбираем базу данных в формате MS Access, задав полный путь к ней, например выбираем демонстрационную базу в каталоге примеров: C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb, входящую в поставку Delphi. При желании проверяем соединение с ней нажатием кнопки Проверить подключение, после чего закрываем все диалоговые окна.

В данном случае процесс установки связи происходит на русском языке, потому что из Delphi вызываются встроенные в русифицированную версию Windows средства обслуживания ADO-технологии.

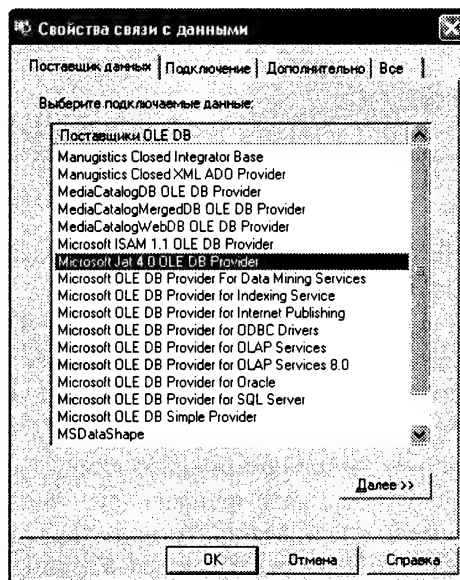


Рис. 5.13. Выбор поставщика данных

Строка соединения сформируется примерно в таком виде:

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Program  
Files\Common Files\Borland Shared\Data\dbdemos.mdb;Persist  
Security Info=False
```

Ее можно увидеть в свойстве `ConnectionString` компонента соединения.

4. Следующий шаг — добавление в проект компонента, ответственного за работу с набором данных. Им будет компонент `TADODataSet`. С объектом `ADOConnection1` он связывается через свойство `Connection`. Однако пока мы не определили, какие же конкретно данные поступят в этот набор. Задать их можно, обратившись к свойству `CommandText` (SQL-команда системе управления базой на формирование набора), — автоматически вызовется небольшой редактор запроса к базе данных. Выберем таблицу `customer` и все поля из нее (звездочка в списке полей `Fields` текущей таблицы) — рис. 5.14.

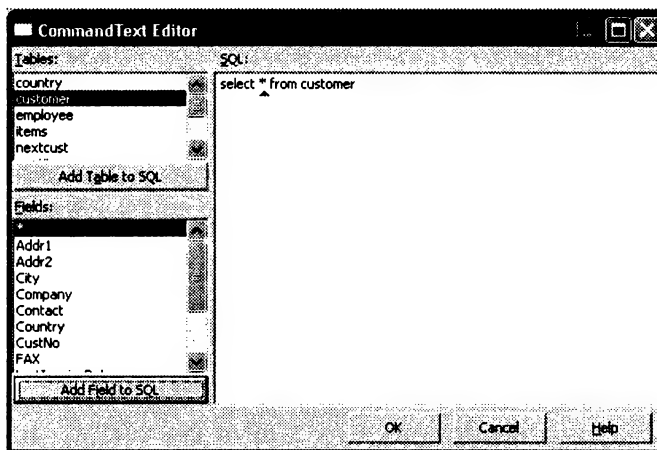


Рис. 5.14. Формирование SQL-запроса к таблице базы данных

5. Нажмем OK. Теперь соединение можно сделать активным — указать значение `True` в свойстве `Active` набора данных.
6. Далее требуется сформировать список полей таблицы `customer`, доступных для обработки. Это делают командой `Fields Editor` из контекстного меню, которая открывает мини-редактор коллекции. С помощью меню (или нажатием клавиш `Ctrl+F`) добавляем в список все поля и закрываем его.

Для визуализации данных с помощью компонентов группы `Data Controls` необходим еще один промежуточный слой — абстрактный источник данных, не зависящий от конкретной технологии работы с наборами данных. Вполне подойдет хорошо знакомый нам компонент `TDataSource` из группы `Data Access`. Его свойство `DataSet` связывается с набором данных ADO, после чего на форму добавляется визуальный компонент группы `Data Controls`

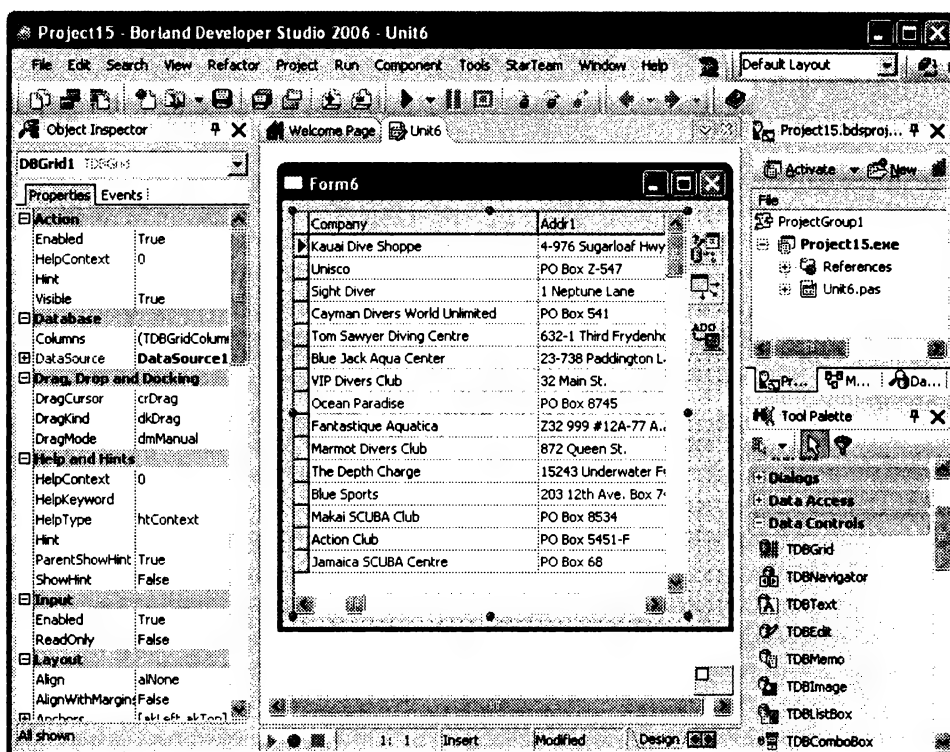


Рис. 5.15. Визуализация табличных данных в Дизайнере

(например, TDBGrid) и связывается с этим новым источником. Данные в таблице показываются сразу в Дизайнере (рис. 5.15).

Для удобства и простоты можно использовать готовые ADO-компоненты, ответственные за представление определенной таблицы (TADOTable; конкретная таблица исходной базы данных задается в свойстве TableName) или результата запроса (TADOQuery; команду запроса придется вводить вручную в свойство SQL).

5.3. Технологии создания многоуровневых приложений баз данных

Мы рассмотрели типичные технологии для создания клиент-серверных систем, предложенные компанией Borland. С их помощью можно создавать и развертывать как локальные решения для работы с базами данных, так и масштабные корпоративные системы. При этом будут соблюдены все требования к целостности и сохранности данных, надежности и устойчивости

работы, поддержке множества пользователей. Следующим принципиальным шагом в развитии архитектуры СУБД считается Web-технология, когда доступ к базе данных происходит из любого браузера обращением по обычному сетевому адресу, что позволяет использовать в качестве клиентских любые цифровые устройства — от обычных персональных компьютеров до мобильных аппаратов. При этом, впрочем, возникает новое промежуточное звено между клиентской программой (браузером) и СУБД. Это Web-сервер, реализующий значительную часть логики системы.

Однако на практике существует еще целый ряд промежуточных технологических задач, потребность в решении которых высока. Так, на крупных предприятиях обычно действует немало разрозненных систем, работающих с различными операционными системами и различными типами СУБД, где постоянно возникает необходимость в стыковке различных приложений и баз данных. Здесь надо выделить два класса задач, которые представляют собой существенное развитие клиент-серверной модели. В свое время они и предвосхитили появление корпоративных Web-систем.

Главными отличиями Web-систем от вышеописанной клиент-серверной архитектуры считаются, во-первых, организация в одном приложении взаимодействия с работающими СУБД различных типов в сетях с разными операционными системами. Во-вторых, создание промежуточных, серверных приложений, которые берут на себя организацию доступа к базе данных, однако выполняются на некотором выделенном компьютере, сервере. Клиентские приложения взаимодействуют не напрямую с СУБД, а с этим серверным модулем, что позволяет существенно снизить нагрузку на сервер баз данных и вынести промежуточную логику обработки таблиц в некое дополнительное серверное звено. Такая программная архитектура называется *многоуровневой* (уровень клиентского приложения — уровни серверных модулей — уровень СУБД) или *многозвенной* (*multitier*), а промежуточные программы относятся к так называемому классу *middleware* (*программное обеспечение промежуточного слоя*). Существуют, в частности, специальные серверы приложений для поддержки программ уровня *middleware*, хотя система Delphi уже включает в себя готовые средства построения многоуровневых систем без привлечения дорогостоящих коммерческих продуктов.

5.3.1. Технология создания многоуровневых BDP-приложений баз данных

Технология создания многоуровневой системы в Delphi выглядит так. Во-первых, надо создать сервер приложений, промежуточное звено между СУБД и клиентскими программами, для чего задействуются компоненты *BdpConnection* и *BdpAdapter* и настраивается связь с СУБД, как уже описывалось. Далее, к проекту добавляется компонент *DataSync*, обеспечивающий взаимодействие набора данных (.NET-класса *DataSet*) с множественными гетерогенными источниками данных. Через посредство этого компонента к серверу

сможет обращаться множество клиентских приложений, и каждому из них компонент **DataSync** обеспечит доступ к нужной таблице некоторой СУБД. Наконец, надо использовать компонент **RemoteServer** (Удаленный сервер), который был специально введен в Delphi для организации взаимодействия клиентских программ с компонентом **DataSync** по одному из стандартных сетевых протоколов.

Пример: создание сервера приложений

1. При создании серверной программы сначала создается заготовка пустого модуля **Windows Forms**. Далее происходит связывание ее с одной из доступных СУБД. Это делается стандартным способом, с помощью **BDP-провайдеров**. Затем готовится набор данных — компонент **DataSet**.
2. Следующий важный шаг — добавление компонента **DataSync** из группы **Borland Data Provider** палитры инструментов. Для его настройки надо вызвать визуальный редактор коллекции поставщиков данных, который данный компонент должен обслуживать. Редактор этот вызывается обращением к свойству **Providers** компонента и нажатием на кнопку **Add** в появившемся окне. В результате к доступному в текущем приложении списку провайдеров добавляется новый элемент (список **Members**) — рис. 5.16.

В правой части редактора выведены свойства текущего поставщика. Так, в свойстве **DataAdapter** указывается доступный в проекте **Vdp-адаптер** (например, **BdpDataAdapter1**). Остальные свойства можно не настраивать, так как они были сгенерированы автоматически при добавлении к проекту связи с СУБД.

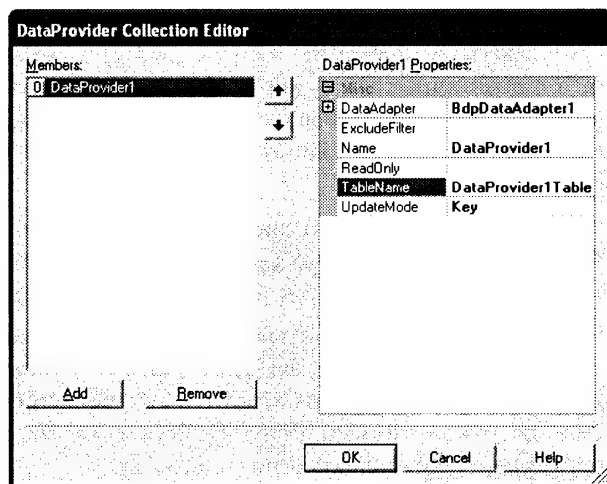


Рис. 5.16. Редактор коллекции поставщиков данных

3. Далее добавим компонент `RemoteServer` и настроим его так: в свойстве `DataSync` укажем текущий объект синхронизации данных (например, `DataSync1`), в свойстве `ChannelType` выберем вид протокола связи с сервером (`Tcp` или `Http`), при необходимости изменим номер порта с 8000 на другой (свойство `Port`). Также сделаем наш сервер активным сразу после его запуска, для чего установим значение `True` свойству `AutoStart`.



Обратите внимание на значение свойства `URI` компонента `RemoteServer`. Именно под таким сетевым именем (по умолчанию `RemoteServer1`) он будет виден в клиентских программах. Чтобы не возникали проблемы с одинаковыми названиями, значение свойства `URI` лучше изменить.

4. Скомпилируем исполнимое приложение и запустим его, только не из среды Delphi, а автономно (например, из Проводника). Это необходимо для отладки клиентской программы. Ведь для ее работы требуется уже настроенный сервер, прослушивающий обращения по заданному порту.

Пример: создание клиентского приложения

Теперь займемся созданием клиентского приложения, которое будет обращаться к северу по указанному нами протоколу.

1. Создадим новое пустое приложение Windows Forms. К нему прежде всего надо добавить компонент, поддерживающий набор данных (`DataSet` из группы `Data Components`), и компонент, визуализирующий этот набор (`DataGrid` из группы `Data Controls`), связав их через свойство `DataSource`.
2. За связь клиентского приложения с сервером будет ответственен компонент `RemoteConnection` (Удаленное соединение) из группы `Borland Data Provider`. Укажем корректный тип протокола связи (значение свойства `ChannelType` должно совпадать со значением этого свойства компонента `RemoteServer` сервера), а также адрес хоста, на котором сервер развернут (свойство `Host`), номер порта (свойство `Port`) и, самое главное, адрес сервера в сети — `URL` (в зависимости от выбранного протокола схема его записи может меняться).

Кроме того, надо убедиться, что название сервера (свойство `URI`, универсальный идентификатор сетевых ресурсов) совпадает с соответствующим значением свойства `URI` компонента `RemoteServer` серверного приложения.

3. Если все эти характеристики настроены правильно, а серверная программа запущена и доступна по указанному адресу, то в свойстве `ProviderType` компонента `RemoteConnection` появится список доступных серверов, поддерживающих технологию `DataSync`. Выберем тот, который соответствует нашему серверному приложению (рис. 5.17).

Таким образом, связь клиентской программы с сервером будет успешно установлена, причем еще на этапе проектирования. Не составляет большо-

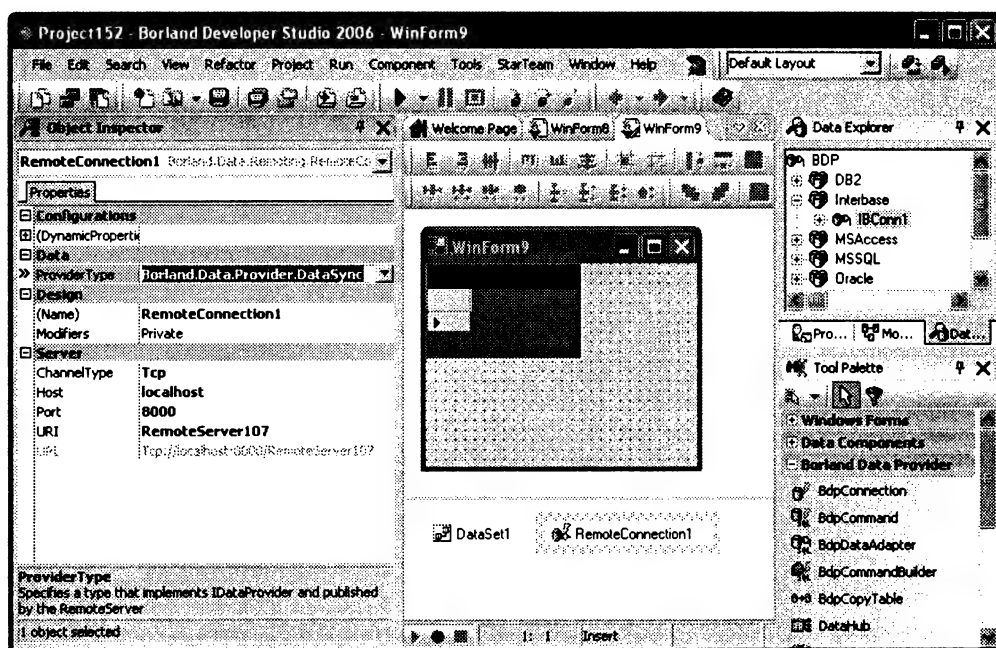


Рис. 5.17. Настраиваем связь приложения с сервером

го труда на ее основе сформировать полноценный пользовательский интерфейс. Для этого достаточно добавить в проект компонент DataHub — прослойку между компонентом RemoteConnection, обеспечивающим связь с сервером, и набором данных DataSet локального приложения. Он играет примерно ту же роль, что и компонент DataSync в сервере. Настраивается этот компонент просто. Со стороны сервера он поддерживает работу через свойство DataPort — в раскрывающемся списке будет показан набор доступных объектов, представляющих текущие URL-соединения. В нашем случае укажем компонент RemoteConnection.

На уровне текущего приложения компонент DataHub связывается с локальным набором данных через свойство DataSet. Если теперь запустить клиентскую программу, то она первоначально предложит в своей форме выбрать нужную для обработки таблицу. Список этих таблиц был сформирован в серверном приложении через компонент DataSync (его свойство Providers — список дистанционно доступных провайдеров данных с привязанными к ним таблицами). При щелчке на названии нужной таблицы она откроется в локальном приложении.

Конкретизировать внешний вид клиентской формы можно, переведя в Дизайнере состояние компонента DataHub в активное состояние (изменив его свойство Active на True) и в свойстве DataMember визуальной таблицы DataGrid указав конкретную таблицу базы данных (рис. 5.18).

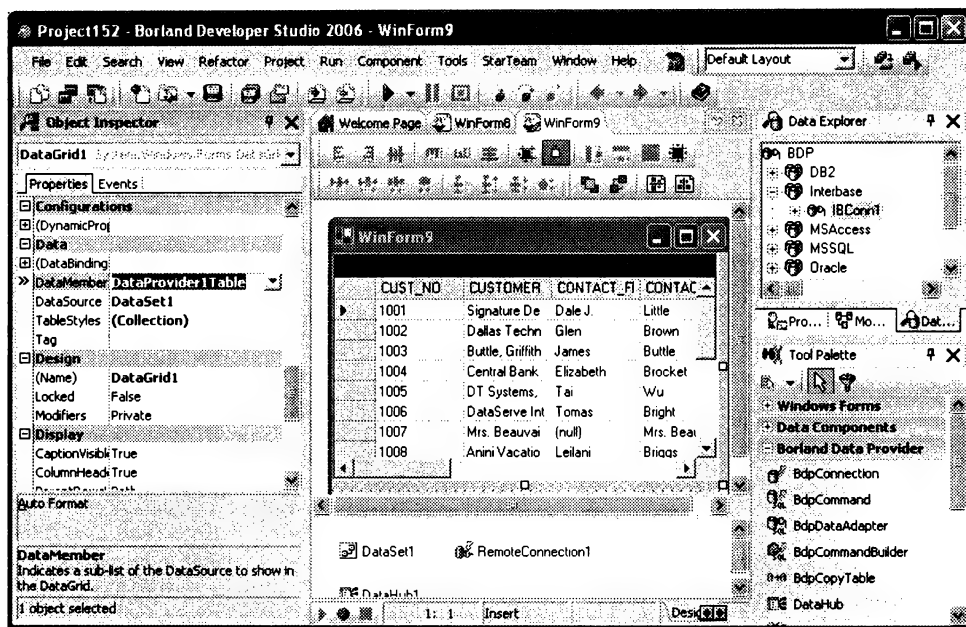


Рис. 5.18. Просмотр удаленной базы данных на этапе проектирования

5.3.2. Создание многоуровневых приложений для платформы .NET с помощью технологии DataSnap

Ограничение вышеизложенной технологии построения многоуровневой системы в том, что она ориентирована только на одну схему обмена данными: по TCP/HTTP-протоколам. Во многих проектах бывает полезно, а порой и необходимо обеспечить взаимодействие клиентов с сервером с помощью самых разных технологических средств: Web-подходов, распределенной технологии Windows DCOM, платформно-независимой технологии CORBA или новой версии платформы .NET. В среде Delphi существует набор компонентов DataSnap, существенно упрощающий необходимые действия.

Допустим, мы хотим создать серверное приложение, работоспособное на платформах Windows 2000/2003/XP, имея ввиду, что клиентские программы будут взаимодействовать с ним как из среды Windows 32, так и из среды .NET.

Пример: создание сервера приложений

1. Сервер создается командой File > New > VCL Forms Application > Delphi for Win32 (Файл > Создать > Приложения VCL Forms > Приложение Delphi Win32).
2. Командой File > New > Other (Файл > Создать > Прочее) и выбором значка Remote Data Module (Модуль удаленного доступа к объекту) из раздела Delphi Projects / Multitier к проекту добавляется модуль, обеспечивающий дистан-

ционную связь сервера с СУБД. В диалоговом окне построения такого модуля, в поле CoClass Name введем название интерфейса этого модуля, доступного в клиентских приложениях, например D2006Test — (рис. 5.19).

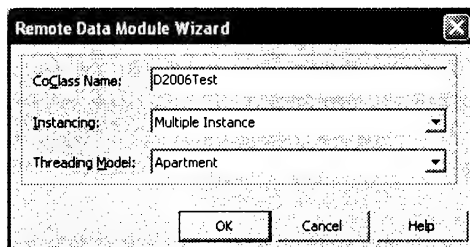


Рис. 5.19. Средство построения модуля удаленного доступа к базе данных

3. Когда модуль будет создан, в него надо внести компоненты связи с СУБД, например TSQLConnection и TSQLTable, которые задают связь с системой Interbase. А в качестве основного компонента, способного взять на себя рутинную функциональность сервера приложений (обслуживание клиентских запросов), выступит компонент TDataSetProvider из группы Data Access палитры инструментов. В его свойстве DataSet надо задать ссылку на компонент TSQLTable, после чего свойству Active последнего надо присвоить значение True, чтобы активизировать сервер (рис. 5.20).

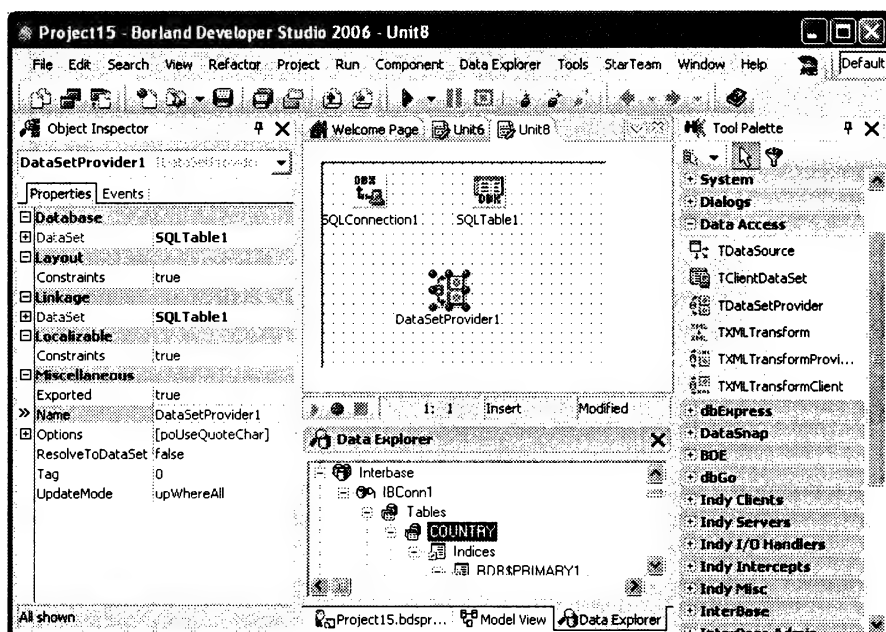


Рис. 5.20. Настройка сервера приложений

Таким образом, сервер приложений создан. Осталось подготовить его для работы на серверном компьютере: он должен постоянно находиться в памяти при обслуживании клиентских программ, как и сервер Bdp-соединений в предыдущем примере. Однако находиться в таком состоянии постоянно не всегда имеет смысл, поэтому сервер приложений перед использованием надо зарегистрировать в той версии Windows, где он будет эксплуатироваться (отличие от предыдущего примера). Для этого сервер запускается с параметром `/regserver` в поле **Parameters** (Параметры) окна **Project Options** (Параметры проекта), которое открывается командой **Run** > **Parameters** (Запуск > Параметры) — рис. 5.21.

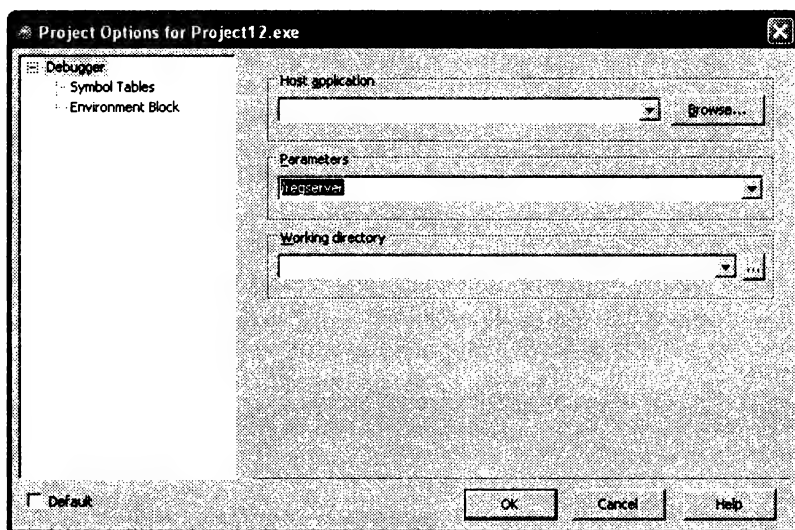


Рис. 5.21. Настраиваем регистрацию сервера при его запуске

Если теперь дать команду **Run** > **Run** (Запуск > Запустить), то сервер запустится, автоматически зарегистрируется в Windows, а потом сразу закроется. После этого строку-параметр запуска надо очистить.



В дальнейшем, когда сервер надо будет исключить из реестра системы, запустите его с параметром `/unregserver`.

Такое создание сервера — типовая для Delphi процедура. А вот возможность формирования клиентских программ в Delphi 2006 существенно расширена.

Пример: доступ к серверу из приложений .NET

1. Создадим пустую программу командой **File** > **New** > **VCL Forms Application** — **Delphi for .NET** (Файл > Создать > Приложения VCL Forms > Delphi для .NET).

2. Для доступа к серверу приложений (он должен быть к этому моменту запущен) поместим на форму компонент `TDCOMConnection` из группы `DataSnap` палитры инструментов. Если обратиться к его свойству `ServerName`, то раскрывающийся список покажет все DCOM-серверы, доступные в текущей операционной системе Windows. Выберем созданный нами ранее `Project1.D2006test`. Наличие установленной связи задается свойством `Connected` — ему надо дать значение `True`. В результате уже на этапе проектирования разработчику становится доступным дистанционный сервис.



Если сервер приложений не был к этому моменту запущен, то после задания значения `True` свойству `Connected` он запустится автоматически: ведь Windows знает его местонахождение, так как он был зарегистрирован в системе.

3. Поместим на форму клиентский (локальный) набор данных (компонент `TClientDataSet` из группы `Data Access` палитры инструментов) — он связывается с компонентом взаимодействия с сервером через свойство `RemoteServer` (будет доступно значение `DCOMConnection1`).
4. Еще один необходимый шаг — выбор дистанционного поставщика данных (один из компонентов `TDataSetProvider` сервера), коих в серверном приложениях может быть множество. Этот выбор осуществляется с помощью

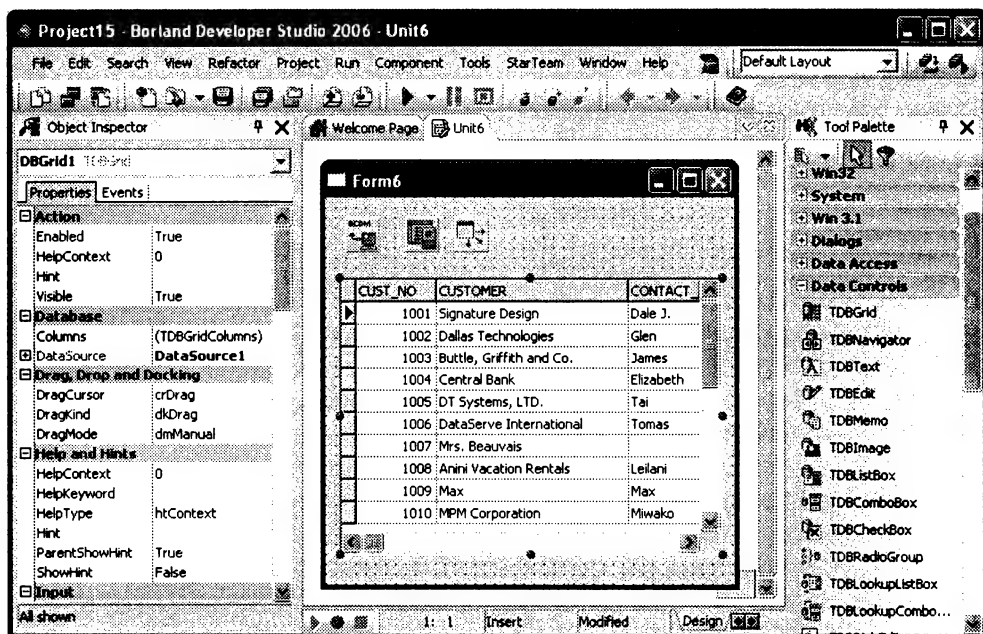


Рис. 5.22. Просмотр таблицы удаленной базы данных на этапе проектирования

свойства `ProviderName`. Если выполняется тестовый пример, то доступным будет, скорее всего, объект `DataSetProvider1`.

5. Затем клиентский набор данных настраивается на обработку списка полей дистанционно получаемой таблицы базы, например с помощью редактора полей (пункт `Fields Editor` контекстного меню). Стыковка набора данных с элементами интерфейса выполняется, как обычно, с помощью компонента `TDataSource`, а их визуализация — с помощью компонента `TDBGrid` и других элементов группы `Data Controls`. Данные, полученные с сервера, будут сразу же показаны в Дизайнере (рис. 5.22).

Схожим образом создается клиентское приложение и для платформы Win32 — в соответствующей группе `DataSnap` палитры инструментов будет доступен компонент `TDCOMConnection`.

Глава 6. Технологии создания Web-приложений

6.1. Технологии создания приложений ASP.NET

Напомним, что технология ASP.NET — это дальнейшее расширение технологии ASP (Active Server Pages), позволяющей создавать сценарии, которые выполняются на Web-сервере Internet Information Services (IIS) и формируют HTML-код, который отсылается клиентскому приложению (браузеру). Подобных технологий немало. Можно упомянуть популярные сценарные языки PHP и Perl, которые распространяются свободно, функционируют на разных платформах и применимы для разных Web-серверов. В свою очередь технологии ASP и ASP.NET созданы в корпорации Microsoft и ориентированы в первую очередь на возможности ее Web-сервера IIS и ее платформ Windows и .NET.

Существенным новшеством технологии ASP.NET явилась ее поддержка языков программирования для платформы .NET. То есть она позволяет разрабатывать сценарии и на языке VB.NET, и на языке C#, и на любом другом языке, поддерживаемом средой выполнения CLR. Реально текст сценария чаще всего представляет собой код HTML (который отправляется клиентскому приложению без изменений) со встроенными в него командами сценария. Выполняясь на Web-сервере, эти команды формируют дополнительный HTML-текст уже в рамках собственной логики: сформированные теги в виде обычных строк просто печатаются в поток стандартного вывода сценария. А новые возможности ASP.NET позволяют включать в сценарий вызовы функций из заранее оттранслированных модулей, что существенно повышает общую производительность решения и гибкость системы: ведь подобные модули как обычные .NET-сборки могут напрямую обращаться ко всем функциональным ресурсам платформы .NET. Кроме того, текст сценария и файл модуля обычно физически разделены, что позволяет независимо редактировать как детали интерфейса, имеющие отношение к HTML-описанию, так и логику, с этими деталями напрямую не связанную. Такой подход позволяет динамически строить результирующий HTML-файл, который

принимается браузером. Можно, например, обращаться на сервере к базам данных, после чего форматировать содержимое таблиц, взятых из базы, по правилам HTML.

Еще одна полезная особенность технологии ASP.NET — готовые средства построения экранных форм на стороне клиента (обычно в браузере), что позволяет проектировать пользовательский интерфейс окна браузера в привычном Дизайнере форм и организовывать дистанционное взаимодействие пользователя из браузера с Web-сервером в виде более привычной модели локального приложения (пользовательский интерфейс и логика обработки его событий).

6.1.1. Новые возможности Delphi по поддержке технологии ASP.NET

Для автоматизации развертывания приложений ASP.NET на Web-сервере в систему Delphi добавлен *Менеджер развертывания (Deployment Manager)*, упрощающий перенос необходимых файлов на удаленный хост.

Набор компонентов DB Web, предназначенных для создания Web-приложений, работающих с базами данных, расширен навигационным интерфейсом и компонентом Navigation Extender, преобразующим Web-элементы управления в элементы навигации по базе данных. Добавлены упрощенные средства создания собственных компонентов DB Web.

Два новых компонента DB Web Sound и DB Web Video расширяют возможности сетевого приложения воспроизведением соответственно потоковой аудио- и видеоинформации (посредством установленного в системе клиента мультимедийного проигрывателя).

Логику формирования HTML-элементов можно использовать стандартную, реализуемую по умолчанию в браузере, а можно создавать ее программно, на сервере. Для этого надо явно пометить HTML-элемент в Дизайнере как запускаемый на сервере (команда контекстного меню Run As Server Control). При этом соответствующий элемент расширяется атрибутом id, значение которого совпадает с его одноименным свойством, и атрибутом runat со значением server. После этого он становится доступным в обработчиках событий формы (Delphi-коде), можно, например, менять его цвет, размеры и так далее.



Отметим также перенос на платформу .NET технологий WebSnap и IntraWeb.

6.1.2. Структура приложения ASP.NET

В рамках проекта Delphi Web-форма физически состоит из файла описания пользовательского интерфейса (расширение .aspx) и файла на языке Паскаль, задающего логику его поведения (расширение .pas). Кроме того, в файл описания .aspx в дополнение к коду HTML включаются серверные команды для создания тегов HTML «на лету», в момент выполнения сценария на сервере.

После компиляции файл .PAS преобразуется в DLL-библиотеку, которая вместе с файлом .ASPX размещается в нужном каталоге Web-сервера (соответствующем некоему виртуальному каталогу), а уже в момент работы (при обращении к Web-форме из браузера) файл .ASPX автоматически транслируется в код .NET, что дает возможность вносить изменения во внешний вид Web-формы простой модификацией исходного кода. Это можно выполнять непосредственно на сервере, без корректировки логики работы (обработки действий пользователя), скрытой в DLL.

6.1.3. Пример создания простого приложения ASP.NET

Рассмотрим простейший пример создания приложения ASP.NET, которое получает в браузере (в двух полях ввода формы) некоторые числа и выводит их сумму после нажатия на кнопку. Отметим, что сложение будет, конечно, выполняться на сервере, но сама форма будет доступна пользователю в его браузере.

1. Создадим новый проект: File > New > ASP.NET Web Application (Файл > Создать > Web-приложение ASP.NET). В поле Name (Имя) укажем название приложения — WebPlus, в поле Location (Размещение) оставим значение по умолчанию, ориентированное на текущий Web-сервер (рис. 6.1).

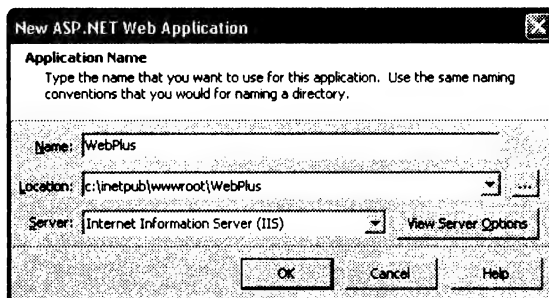


Рис. 6.1. Подготовка к созданию проекта

2. После нажатия на кнопку OK среда Delphi перейдет в режим проектирования Web-формы. Проектирование выполняется способом, аналогичным способу работы с обычным дизайнером форм Windows.

На палитре инструментов выберем раздел Web Controls (элементы Web-управления), после чего поместим на форму два поля ввода edit1 и edit2 (компонент TextBox), поле-надпись label1 (компонент Label) и кнопку button1 (компонент Button). Названия объектов задаются в свойстве ID. В полях ввода укажем нули в качестве начальных значений (свойство Text). Знак «+» создан с помощью поля Label (рис. 6.2).

3. Обработчик нажатия на кнопку тоже формируется стандартным способом. Достаточно щелкнуть на этой кнопке в дизайнера Web-формы — в редакторе появится код, сгенерированный автоматически.

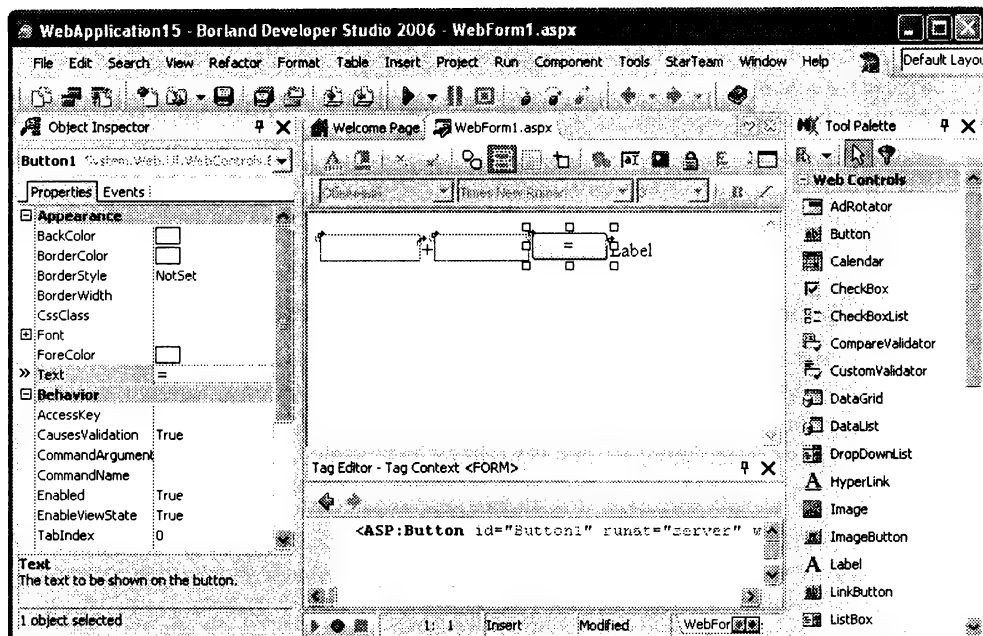


Рис. 6.2. Проектируем Web-форму

```

procedure TWebForm1.Button1_Click(sender: System.Object;
e: System.EventArgs);
begin

```

```

end;

```

Укажем в нем оператор сложения и вывода результата:

```

procedure TWebForm1.Button1_Click(sender: System.Object;
e: System.EventArgs);
begin
    label1.Text := IntToStr( StrToInt(edit1.Text) +
StrToInt(edit2.Text) );
end;

```

Чтобы этот код, выполненный в духе прежних версий Delphi, работал, необходимо подключить модуль SysUtils, в котором хранятся общеизвестные функции преобразования типов:

```

implementation uses SysUtils;

```

Теперь, в принципе, наш проект готов для запуска. Однако мы создаем не приложение Windows, а модуль, который должен выполняться на Web-сервере, поэтому напомним, что для поддержки его работы необходимо, чтобы на компьютере был установлен какой-либо Web-сервер, обеспечивающий

работу сценариев ASP.NET. Проще всего, конечно, использовать стандартный сервер Microsoft IIS. Если он настроен корректно, то достаточно дать команду главного меню Run ➤ Run (Запуск ➤ Запустить), все модули соберутся и приложение запустится. Введем в поля формы браузера значения, нажмем кнопку и получим результат (рис. 6.3).

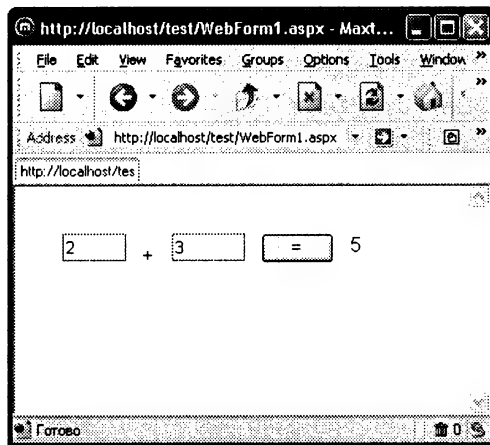


Рис. 6.3. Так работает Web-калькулятор

По данной технологии можно создавать весьма и весьма сложные пользовательские интерфейсы, задействуя всю мощь компонентов группы Web Controls и сочетая ее с неограниченными возможностями программируемой логики языка Delphi.

6.1.4. Работа с редактором кода HTML

При проектировании Web-формы мы пользовались элементами управления категории Web Controls. При этом в исходный текст файла .ASPX вставлялись теги ASP.NET. Часто бывает необходимо использовать стандартные теги HTML для организации дизайна всей страницы и построения стандартных форм ввода. Набор элементов управления, задаваемых тегами HTML, находится в палитре инструментов в группе HTML Elements (рис. 6.4).

HTML-код текущего выбранного элемента представляется в нижнем окне, под Дизайнером. Его можно откорректировать, при этом соответствующие изменения занесутся и в основной файл .ASPX. При модификации исходного текста автоматически будет модифицироваться и внешний вид соответствующего элемента в Дизайнере. При этом разработчику доступны все основные возможности редактора: автозавершение кода, шабло-

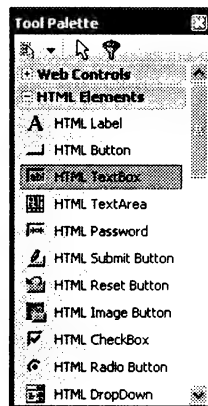


Рис. 6.4. Элементы управления HTML

ны, подсказки. Настройки редактора и Дизайнера HTML-тегов доступны по команде Tools > Options > HTML/ASP.NET Options (Сервис > Параметры > Параметры HTML/ASP.NET).

6.2. Технологии ASP.NET по работе с базами данных

6.2.1. Технология стандартного соединения приложения ASP.NET с базами данных

Основное предназначение технологии ASP.NET — это, конечно, создание Web-систем, работающих с базами данных. Простые приложения, выполняющие арифметические расчеты или реализующие другую несложную логику, можно создавать с помощью более простых технологий (например, на языке PHP или даже с применением языков сценариев наподобие JScript, исполняющихся непосредственно в браузере клиента). Гораздо актуальнее решения, в которых сохраняется информация о каждом пользователе, доступны различные справочные каталоги, возможно обращение к корпоративным данным. Для этого серверный код надо связать с СУБД, расположенной либо на компьютере с Web-сервером, либо на другом компьютере, соединенном с этим сервером (обычно по локальной сети). В любом случае об их местонахождении обычному пользователю заботиться не надо. Он запускает браузер, вводит обычный адрес, открывает Web-страницу, заполняет форму и в ответ на свой запрос получает нужные сведения из удаленной базы данных.

Никаких принципиальных отличий от вышерассмотренного подхода не будет и в данном подходе. DLL-библиотека, сформированная вместе с кодом .ASPX, может обращаться к любым СУБД, находящимся в пределах ее досягаемости — достаточно встроить в нее средства взаимодействия с базами данных.

Пример: заготовка серверного приложения

1. Создадим пустое приложение ASP.NET, как было описано выше. Далее разместим на форме компоненты `BdpConnection` и `BdpDataAdapter` и настроим их на связь, например, с таблицей `CUSTOMER` базы данных `Interbase`, которая использовалась ранее при описании технологий BDP.NET.
2. Для доступа к соответствующему набору данных потребуется также компонент `DataSet`. Выделим в проекте объект `BdpDataAdapter`, в его контекстном меню выберем пункт `Generate Typed Dataset` (Генерировать типизированный набор данных), после чего откроется диалоговое окно `Generate Dataset` (рис. 6.5).

Как видно, здесь предлагается быстрое создание и добавление к проекту готового набора данных `DataSet1`, настроенного на таблицу `CUSTOMER`. Нажмем кнопку OK, и в проекте появится новый объект `dataSet1`.

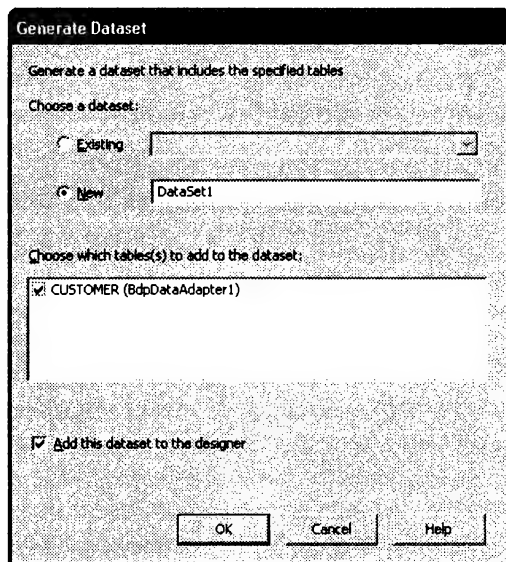


Рис. 6.5. Средство автоматического создания набора данных

3. Для взаимодействия с доступным теперь набором данных задействуем компонент DataGrid из группы Web Controls (рис. 6.6).

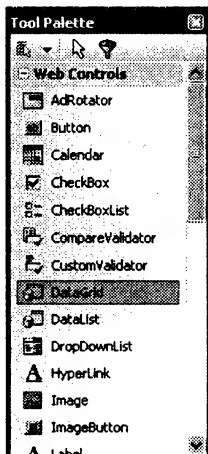


Рис. 6.6. Выбор компонента Web Controls

Разместим этот компонент на видном месте формы и свяжем его с набором данных `dataSet1` через свойство `DataSource`. В свойстве `DataMember` выберем ту таблицу, которую хотим редактировать в браузере клиента (пред-

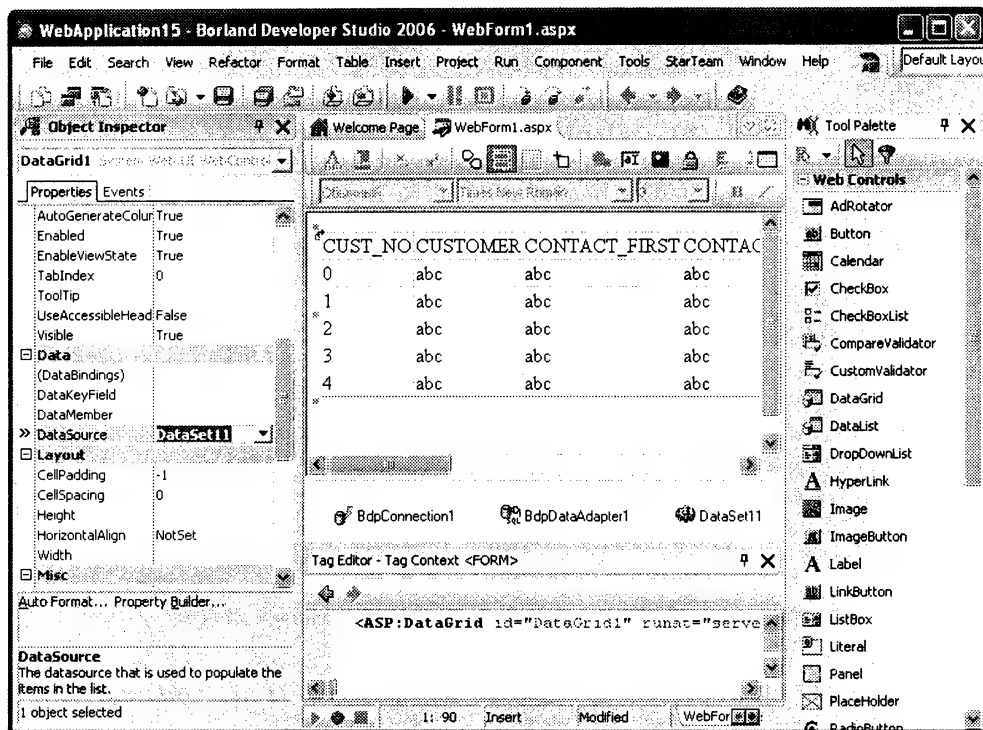


Рис. 6.7. Структура связанной таблицы в Дизайнере форм

варительно активизируем объект `BdpDataAdapter`, задав его свойству `Active` значение `true`). В Дизайнере сразу отобразится структура таблицы (названия полей) — рис. 6.7.

Пример: связывание данных

По умолчанию концепция применения технологии ASP.NET требует, чтобы в программном коде Web-приложения выполнялось явное связывание пользовательского элемента управления с источником данных. Эта связь должна создаваться на этапе компиляции. Реализуется это требование следующим образом. Программист создает обработчик события `Load` (Загрузка) формы и назначает для него стандартное значение `Page_Load` (оно и предназначено для задач связывания).

1. Выделим в Инспекторе объектов Web-форму (объект `TWebForm1`), на вкладке `Events` (События) найдем событие `Load` и установим для него значение `Page_Load` (рис. 6.8).
2. Дважды щелкнем на этой строке и в редакторе запрограммируем обработку данного события.

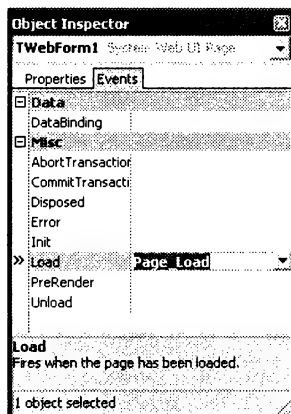


Рис. 6.8. Выбираем перехватываемое событие Page_Load

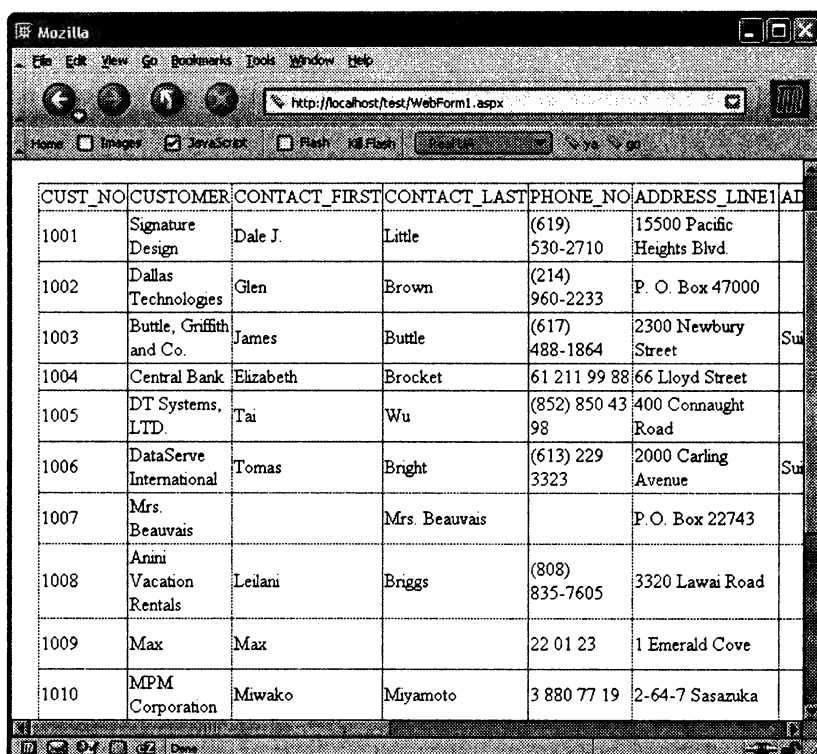
```
procedure TWebForm1.Page_Load(sender: System.Object;  
e: System.EventArgs);  
begin  
    Self.DataGrid1.DataBind;  
end;
```

Здесь вызывается метод `DataBind`, который определяет связь между серверной логикой (формированием HTML-кода таблицы, которая, уже со встроенными значениями, взятыми из базы данных, будет передана клиентскому приложению) и источником данных (в нашем случае адаптером BDP.NET). Предварительно требуется, чтобы объект, вызывающий метод `DataBind`, был настроен на источник данных (объект `dataSet1`) через свое свойство `DataSource`. Мы сделали это в Дизайнере, но это можно сделать и в коде.

```
Self.DataGrid1.DataSource := DataSet1;  
Self.DataGrid1.DataBind;
```

Остановимся на вопросе связывания более подробно. В среде .NET Framework существуют разнообразные формы поддержки связывания данных — соединения набора данных с элементами пользовательского интерфейса, а также с различными внутренними структурами (например, связывание со списком массивов). Метод `DataBind`, входящий в класс `Control` среды .NET (этот класс определяет базовый элемент управления), может быть вызван как для всей формы, за счет чего все наборы данных свяжутся со всеми элементами управления, так и для конкретного элемента. А в качестве набора данных может выступать не только таблица реляционной СУБД, но и любой другой источник.

3. Теперь приложение можно запустить и получить в браузере доступ ко всей таблице удаленной базы данных (рис. 6.9).



CUST_NO	CUSTOMER	CONTACT_FIRST	CONTACT_LAST	PHONE_NO	ADDRESS_LINE1	ADDRESS_LINE2
1001	Signature Design	Dale J.	Little	(619) 530-2710	15500 Pacific Heights Blvd.	
1002	Dallas Technologies	Glen	Brown	(214) 960-2233	P. O. Box 47000	
1003	Buttle, Griffith and Co.	James	Buttle	(617) 488-1864	2300 Newbury Street	Su
1004	Central Bank	Elizabeth	Brocket	61 211 99 88	66 Lloyd Street	
1005	DT Systems, LTD.	Tai	Wu	(852) 850 43 98	400 Connaught Road	
1006	DataServe International	Tomas	Bright	(613) 229 3323	2000 Carling Avenue	Su
1007	Mrs. Beauvais		Mrs. Beauvais		P.O. Box 22743	
1008	Anini Vacation Rentals	Leilani	Briggs	(808) 835-7605	3320 Lawai Road	
1009	Max	Max		22 01 23	1 Emerald Cove	
1010	MPM Corporation	Miwako	Miyamoto	3 880 77 19	2-64-7 Sasazuka	

Рис. 6.9. Полный доступ из браузера к удаленной базе

При создании Web-приложений существует одна тонкая особенность, которая касается как минимум СУБД Interbase. В процессе проектирования формы в Проводнике данных соответствующее соединение (в частности, компонент `VdpConnection`) можно настраивать через локальные координаты физической базы данных, с указанием полного пути к ней.

На рис. 6.10 в первой строке (Database) задан прямой путь к файлу базы Interbase `employee.gdb`. Следует иметь в виду, что эта настройка действует лишь в пределах среды Delphi, а когда начнется отладка приложения, то при запросе к нему из браузера возникнет ошибка невозможности подсоединения к базе. Это связано с тем, что модуль, выполняющийся на Web-сервере, обращается к базе данных по сетевому протоколу, и даже если эта база находится на одной с ним машине, доступ к ней все равно происходит по стандартным IP-протоколам. Поэтому в конкретном проекте в начале записи пути доступа к физическому файлу надо обязательно указать либо название сетевой машины, на которой установлена СУБД, либо ее IP-адрес. На этапе отладки на одном компьютере такая запись может быть выполнена так.

127.0.0.1:C:\tmp\employee.gdb

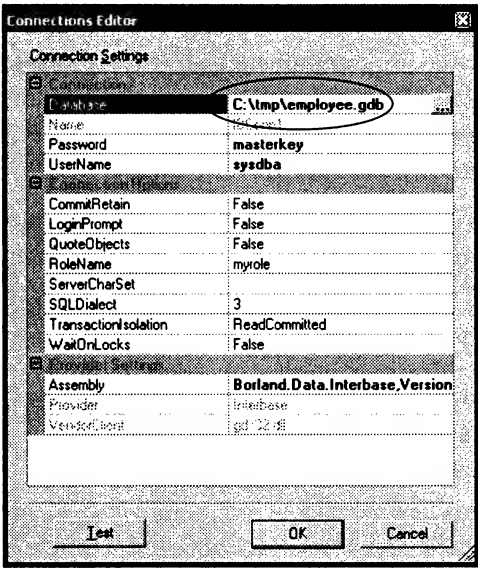


Рис. 6.10. Указанный путь доступа к базе данных актуален только на этапе отладки

За IP-адресом 127.0.0.1 локального компьютера обязательно следует двоеточие, и лишь затем — путь к базе. Цифровой адрес можно заменить, например, на localhost, хотя и не всегда (рис. 6.11).

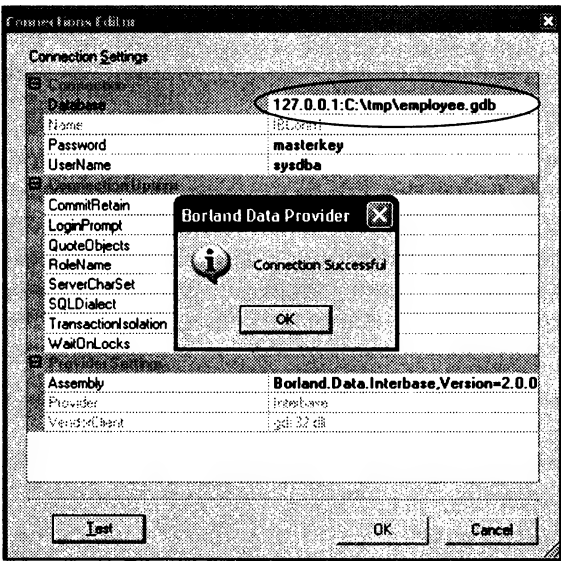


Рис. 6.11. Правильная форма записи адреса базы данных в сети

6.2.2. Технология DB Web

В системе Delphi существует технология DB Web, которая еще больше упрощает процесс создания Web-приложений, взаимодействующих с базами данных. Набор компонентов DB Web даже не требует явной реализации процесса связывания, поскольку выполняет все нужные действия автоматически.

Пример: связь с базами данных с помощью технологии DB Web

1. Выполним вышеописанные шаги по созданию приложения ASP.NET до момента, когда мы создали работающее соединение с СУБД, но еще не разместили на форме пользовательский компонент DataGrid. Вместо него воспользуемся компонентом DBWebDataSource (Источник данных DB Web) из группы DB Web палитры компонентов (рис. 6.12).
2. Свяжем этот компонент с существующим набором данных dataSet1 через свойство DataSource (обратите внимание, что можно сразу выбрать и конкретную таблицу, CUSTOMER). Таким образом компонент DBWebDataSource служит своеобразным переходным звеном между компонентами BDP.NET и объектами интерфейса DB Web.
3. Добавим на форму компонент DBWebGrid, практически аналогичный по предназначению компоненту DataGrid. В его свойстве DBDataSource выберем только что созданный источник DBWebDataSource1, а в свойстве TableName теперь укажем название доступной таблицы CUSTOMER.

На форме сразу же появится содержимое данной таблицы (рис. 6.13). Напомним, что при работе с элементами набора Web Controls данные в Дизайнере сразу не показывались, так как их предварительно надо было связать с этими элементами с помощью метода DataBind, причем непосредственно в процессе работы программы.

4. Теперь достаточно скомпилировать и установить данное приложение на Web-сервере — и к нему можно будет обращаться из браузера. Работоспособная программа (Web-модуль) при этом будет получена без какого-либо ручного кодирования. Не возникает необходимости явного задания связей между источниками данных и элементами пользовательского интерфейса, а самое главное — на форме достаточно одного компонента-источника DBWebDataSource, к которому могут обращаться любые другие компоненты панели DB Web, связывание которых с источ-

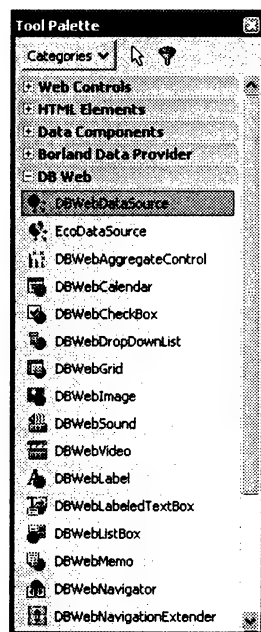


Рис. 6.12. Выбор компонента DBWeb DataSource

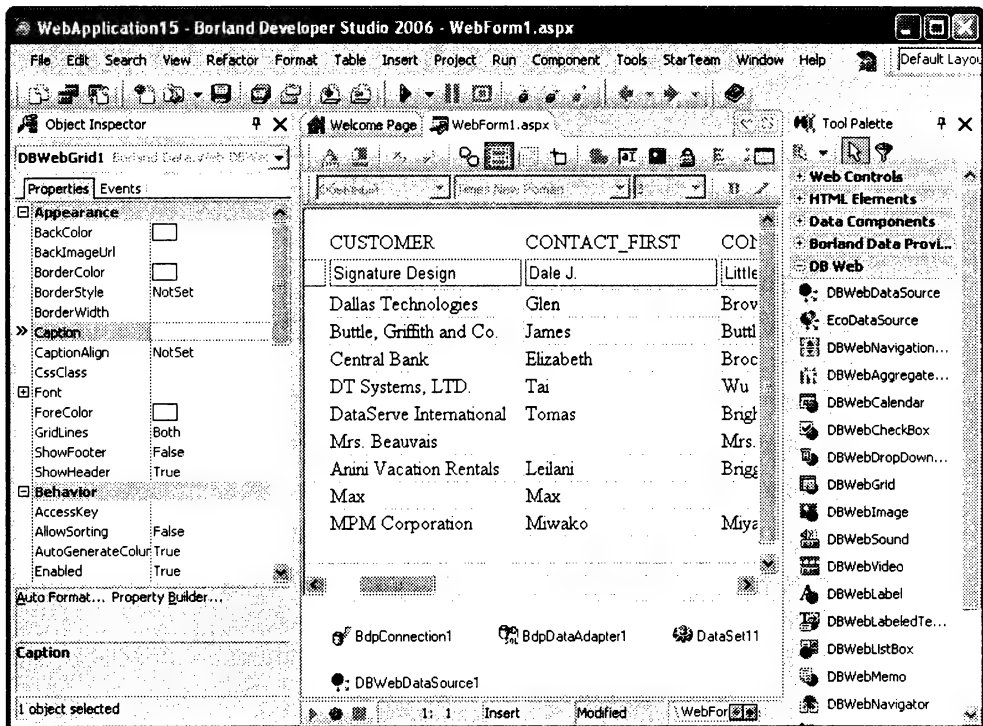


Рис. 6.13. Доступ к таблице баз данных на этапе проектирования

ником данных выполнится автоматически. Кроме того, элементы управления DB Web совместимы с поставщиками данных ASP.NET и BDP.NET. Дополнительно отметим, что использованные нами компоненты обладают встроенными средствами прокручивания набора данных в окне браузера и средствами редактирования значений в текущей строке!

Замечание: DB Web и XML

Еще одна полезная возможность набора компонентов DB Web — их способность работать с XML-файлами как с источниками данных. Такие файлы представляют собой обычный текст с тегами разметки, структурирующими содержимое не только в реляционном, но и, например, в иерархическом виде. При этом надо учитывать, что полная открытость XML-файлов делает их беззащитными от внешнего просмотра, поэтому вряд ли разумно хранить в них какую-либо конфиденциальную информацию. И конечно, если сервер СУБД выполняет множество дополнительных функций по ведению базы данных (поддерживает транзакции, блокирует при необходимости таблицу или запись от конфликтных изменений, сохраняет ее целостность, выполняет различные действия по оптимизации запросов), то работа с XML-файла-

ми таких возможностей не даст. Лучше всего задействовать XML-файлы в качестве своеобразной небольшой локальной базы данных, сведения в которой хранятся не реляционно, а в виде иерархических структур. Чаще всего потребность в обработке XML-файлов востребована в интеграционных приложениях, когда данные из одних баз и систем передаются в другие и нуждаются во временном хранилище, в идеале — имеющем иерархическую структуру.

Пример: сохранение данных в XML-файле

Никаких особых требований для использования XML-файлов в качестве источников данных в рамках технологии DB Web не существует. Рассмотрим сначала, как выгрузить существующий набор данных в XML-файл.

1. Описанным выше способом создадим приложение ASP.NET.
2. Подготовим компоненты связи BDP.NET.
3. Поместим на форму уже знакомый компонент DBWebDataSource.
4. Выберем XML-файл и в его свойстве XMLFileName укажем пока несуществующее имя файла.
5. Теперь достаточно запустить приложение и соответствующий файл будет создан.

Итак, если требуется перевести некоторый набор данных в XML-формат (сохранить в виде XML-файла), то применяется вышеописанная схема, когда компонент DataSet связывается с существующим адаптером данных BDP.NET (он размещается на форме и настраивается заранее, после чего переводится в активное состояние). А в качестве XML-файла в свойстве XMLFileName источника данных DBWebDataSource указывается несуществующий файл, и после запуска такого приложения этот файл будет создан, а в него записана размеченная в соответствии с требованиями спецификации XML информация из набора данных, заданного компонентом DataSet.

Основной недостаток использования XML-файлов в качестве «базы данных», как уже говорилось, заключается в том, что организовать корректную работу множества пользователей с одним файлом практически невозможно. Как только одна из клиентских программ начинает модифицировать такой файл-базу, все остальные пользователи будут либо ждать окончания этого процесса, либо пытаться одновременно занести в файл противоречивые данные, модифицированные разными клиентами. При работе с компонентом-источником данных DB Web можно явно назначить режим поддержки множества копий рабочего XML-файла, для чего в свойство UseUniqueFileName надо занести значение true. Однако при этом необходимо аккуратно задать права доступа дистанционных клиентских приложений к каталогу с этим файлом (указать в Windows права записи в этот общедоступный каталог).

В результате для каждого клиентского подключения к XML-файлу, указанному в свойстве `XMLFileName`, во избежание противоречий будет формироваться временный XML-файл с уникальным именем для «персонального» использования. Однако анализ и синхронизацию содержимого этих файлов разработчику придется выполнять самому. Например, создавать отдельные утилиты, которые в плановом режиме будут запускаться на сервере и объединять все данные, поступившие из клиентских XML-файлов.

Задача эта не тривиальная, поэтому представляется оптимальным эксплуатировать XML-файлы прежде всего в качестве источников данных в режиме «только для чтения» (например, для последующей загрузки в полноценную реляционную СУБД).

Пример: загрузка данных из XML-файла

Загрузка набора данных из XML-файла происходит следующим образом.

1. Создадим пустое Web-приложение ASP.NET.
2. Разместим на форме компонент `DataSet` из раздела `Data Components` палитры инструментов, а также компонент `DBWebDataSource`.
3. В свойстве `XMLFileName` последнего источника данных зададим существующий XML-файл, а в свойстве `DataSource` — набор данных (`DataSet1`).
4. Далее добавим какой-либо из визуальных компонентов `DB Web`, например `DBWebGrid`, и свяжем его через свойство `DataSource` с источником данных `DBWebDataSource`. В результате он отобразит содержимое XML-файла как содержимое таблицы данных.

6.2.3. Рекомендации по настройке и отладке приложений ASP.NET

По умолчанию Web-сервер позволяет загружать автоматически некоторое количество файлов при обращении к сайту. Известный пример — файл `index.html`. Имена этих файлов можно указать в окне консоли `Internet Information Services`. Выбрав пункт `Веб-узел` по умолчанию, в его контекстном меню надо дать команду `Свойства` и перейти на вкладку `Документы` — там приведен список файлов, поддерживаемых по умолчанию. С помощью кнопки `Добавить` можно указать новый файл, например `WebForm1.aspx`. Это удобно, когда виртуальные каталоги на Web-сервере создаются автоматически при открытии соответствующих проектов в Delphi, а в них создается только файл `WebForm1.aspx`. Если к такому каталогу обратиться, опустив полное название документа, то возникнет ошибка, если этот документ не занесен в список умолчаний (рис. 6.14).

Замечание: сервер Cassini

Если в распоряжении разработчика имеется только один компьютер, то для отладки приложений ASP.NET весьма удобно воспользоваться открытым

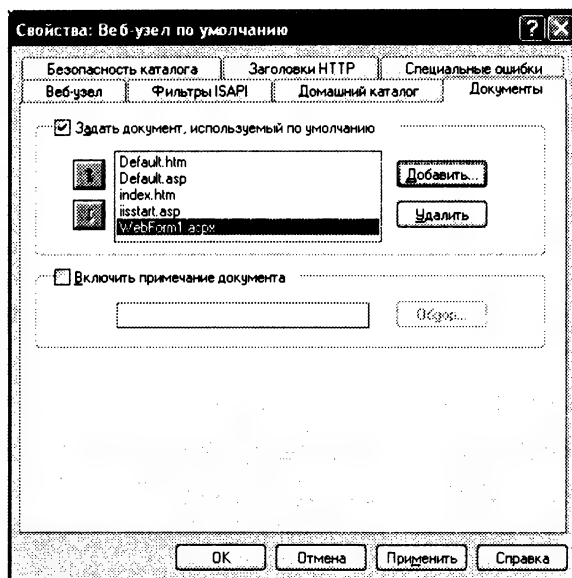


Рис. 6.14. Формирование списка документов, загружаемых по умолчанию

сервером Cassini, который входит в поставку системы Delphi. Он хорош тем, что требует минимальных настроек, чем отличается от сервера IIS, который начинает работать быстро и устойчиво только после изрядных конфигурационных изменений. Отличительная особенность Cassini — поддержка разработки именно на локальном компьютере: для полноценного серверного использования необходимо вносить изменения в его исходные тексты. Впрочем, вряд ли он может соревноваться с другими общедоступными Web-серверами, например с Apache, а вот для разработки и отладки подходит как нельзя лучше.

В принципе, последнюю версию Cassini можно загрузить с сайта разработчиков www.asp.net/Projects/Cassini/, причем в поставку может входить и файл .EXE, но по различным причинам им лучше не пользоваться (почти всегда он оказывается неработоспособным и не может разобраться с портами 80 и 8080). Рекомендуется полностью собрать новую версию из исходных текстов. Написан Cassini на языке C#, поэтому необходимо, чтобы в системной переменной PATH был прописан полный путь к компилятору C#, входящему в поставку .NET Framework. Скорее всего переменную PATH надо будет дополнить строкой наподобие:

```
c:\Windows\Microsoft.NET\Framework\v1.1.4322\
```

Сам компилятор командной строки называется `csc.exe` (*C Sharp Compiler*). Чтобы собрать сервер Cassini из исходных текстов (они расположены в каталоге примеров `Borland\BDS4.0\Demos\Cassini`), надо запустить командный файл

build.bat, после чего в этом каталоге появится файл CassiniWebServer.exe. Когда при создании приложения ASP.NET в качестве Web-сервера будет впервые указан сервер Cassini, система Delphi запросит его местонахождение (файл CassiniWebServer.exe) и предложит выполнить мелкие настройки (указать рабочий порт, например 8080; установить виртуальный каталог для хранения создаваемого Web-модуля). Уточнение всех этих особенностей можно отложить до момента создания проекта.

6.3. Технологии создания Web-служб (Web Services)

6.3.1. Принцип быстрой разработки Web-служб

Концепция Web-служб сегодня по праву считается наиболее перспективным направлением развития сетевого компонентного подхода. Web-служба представляет собой программный интерфейс, к функциям которого можно обращаться дистанционными запросами в стандартной форме по локальной и глобальной сетям. Компоненты, выполненные в классических моделях, остаются довольно сложными в развертывании, программирование обращений к их функциям также не всегда легко выполняется, а поддерживаемые интерфейсы зачастую несовместимы друг с другом. Но самый главный недостаток действующих компонентных моделей заключается в том, что они требуют для своего функционирования определенных архитектур, сред, операционных систем. Поэтому разворачивать и сопровождать на предприятии одновременно несколько платформ (например, Java, CORBA, .NET) весьма накладно и трудоемко.

Web-служба является сетевым развитием компонентной концепции. Процесс использования любой Web-службы не связан с формой ее реализации, то есть работать такая служба может на Unix-сервере или Windows-машине, в мобильной Java-среде или на мэйнфрейме, но в каждом случае для обращения к ней достаточно иметь обычный браузер, действующий на произвольной программно-аппаратной платформе. Это позволяет, в частности, состыковывать множество внутренних и внешних приложений предприятия, не создавая для этого дополнительную инфраструктуру.

Весь процесс взаимодействия клиентской программы с Web-службой происходит, как правило, на основе высокоуровневого протокола Simple Object Access Protocol (SOAP), который, в свою очередь, основан на текстовом языке разметки XML, что позволяет использовать для работы с Web-службами стандартный текстовый протокол HTTP. Если обращение к Web-службе должно происходить программно, в автоматическом режиме, то первым делом она должна предоставлять полное описание реализуемого ею программного интерфейса. Оно должно содержать перечень доступных для вызова методов с указанием их параметров и типов, а также формат возвращаемого значения. Для организации такого взаимодействия предназначен Web Services Description Language (WSDL) — язык описания Web-служб.

С его помощью программа-клиент автоматически выясняет подходящий способ обращения к конкретной Web-службе. А когда происходит вызов конкретного метода из интерфейса Web-службы, та обращается к Web-серверу, на котором работает (сервер должен поддерживать соответствующие технологии Web Services). После этого автоматически создается объект, реализующий интерфейс службы, выполняется нужный метод этого объекта, а результат возвращается сервером клиенту по SOAP-протоколу.

6.3.2. Структура Web-службы

В ходе создания Web-службы компилятор Delphi автоматически сгенерирует ряд файлов, одни из которых будут в дальнейшем модифицироваться автоматически, а другие будут хранить код, подготовленный вручную. Автоматически создаваемый файл с расширением .ASMX задает связь между Web-службой и ее адресом URL, по которому она будет доступна конечным пользователям, и определяет класс, экземпляр которого будет создан при обращении к службе для выполнения ее функций. Кроме того, в этом файле, как и в сценарных файлах ASP.NET, можно расположить и программный код (например, на языке C#), определяющий логику работы службы, однако делать это на языке Delphi пока невозможно.

Файл проекта .PAS хранит непосредственный код обработки пользовательских запросов — основную прикладную логику, реализацию необходимых разработчику функций интерфейса Web-службы.

Файл .WSDL хранит стандартное XML-описание Web-службы (ее открытый интерфейс), которое будет передаваться приложениям, запрашивающим перечень методов, доступных для дистанционного вызова.

Результирующий файл Web-службы (в формате Windows) выполнен в формате динамически загружаемой библиотеки и имеет расширение .DLL.

6.3.3. Технология создания простой Web-службы

Рассмотрим создание несложной Web-службы, умножающей два полученных дробных числа.

1. Новый проект создается командой главного меню File ➤ New ➤ Other (Файл ➤ Создать ➤ Другое), на вкладке Delphi for .NET Projects (Проекты Delphi.NET) выбирается значок ASP.NET Web Service Application (Приложение Web-службы ASP.NET) — рис. 6.15.

После нажатия кнопки ОК, как обычно для Web-приложений, укажем название проекта, например WebService1, и каталог для его хранения. В каталоге будет создана пара файлов, например WebService1.asmx и WebService1.pas.

2. В редакторе Delphi перейдем к файлу WebService1.pas. По умолчанию в нем автоматически создан метод-пример HelloWorld, который войдет в открытый интерфейс нашей службы. Пока он скрыт в комментариях.

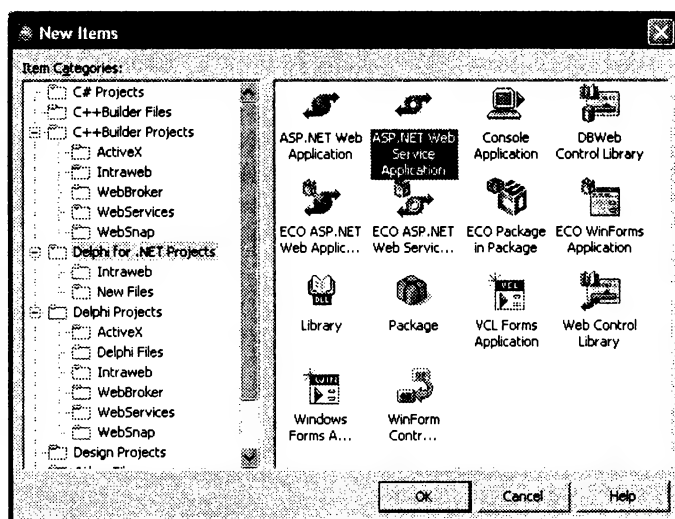


Рис. 6.15. Выбор типа проекта

В интерфейсе:

```
TWebService1 = class(System.Web.Services.WebService)
...
public
  constructor Create;
  (*
  // Sample Web Service Method
  [WebMethod]
  function HelloWorld: string;
  *)
end;
```

В реализации:

```
// Sample Web Service Method
// The following method is provided to allow for testing a
new web service.
(*
function TWebService1.HelloWorld: string;
begin
  Result := 'Hello World';
end;
*)
```

3. Снимем комментарии с обеих частей, после чего у нас окажется готовая заготовка Web-службы. После компиляции в целях тестирования к ней можно обращаться из браузера, например, так.

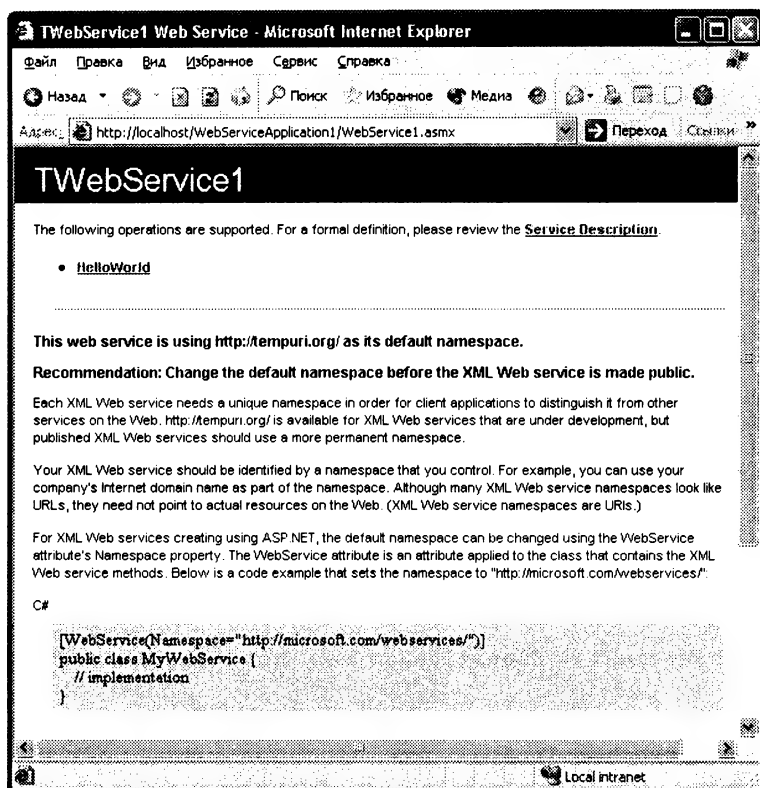


Рис. 6.16. Web-служба в действии

`http://localhost/WebServiceApplication1/WebService1.asmx`

Здесь считается, что проект размещен в виртуальном каталоге `WebServiceApplication1` Web-сервера. Отклик браузера, если настройки сервера выполнены правильно, будет таким, как показано на рисунке 6.16.

В этом окне имеется все необходимое для тестирования нашей службы. Мы рассмотрим эти возможности далее.

- Добавим к нашей программе новый метод умножения. Назовем его `WebMul`. Два его параметра будут представлять дробные числа, а возвращать метод будет дробное значение, равное их произведению. Перед описанием заголовка метода в классе необходимо указать атрибут `WebMethod`, который сообщает компилятору, что следующий метод может использоваться через открытый интерфейс Web-службы:

```
TWebService1 = class(System.Web.Services.WebService)
...

```

```

public
  constructor Create;

  // Sample Web Service Method
  [WebMethod]
  function HelloWorld: string;

  [WebMethod]
  function WebMul(x,y: Double): Double;

end;

```

Реализация:

```

function TWebService1.WebMul(x, y: Double): Double;
begin
  Result := x * y ;
end;

```

- Обратившись из браузера по указанному выше адресу, рядом со ссылкой HelloWorld мы увидим новую ссылку WebMul. Такая удобная возможность интерактивного просмотра интерфейса Web-службы реализуется соответствующими модулями Web-сервера, ответственными за поддержку протокола SOAP и Web-служб. При щелчке на ссылке WebMul откроется окно проверки метода WebMul с автоматически сгенерированной тестовой формой и наглядными полями, совпадающими по названию с именами параметров тестируемого метода. В нижней его части будут подробно описаны форматы SOAP-запроса и SOAP-ответа, оба в формате XML.

```

POST /WebServiceApplication1/WebService1.asmx
HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/WebMul"

```

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <WebMul xmlns="http://tempuri.org/">
      <x>double</x>
      <y>double</y>
    </WebMul>
  </soap:Body>

```



```

</soap:Envelope>

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <WebMulResponse xmlns="http://tempuri.org/">
      <WebMulResult>double</WebMulResult>
    </WebMulResponse>
  </soap:Body>
</soap:Envelope>

```

Видно, что параметры *x* и *y* выступают как названия тегов, а их типы — как значения. По схожей схеме формируется и возвращаемое значение *WebMulResult*.

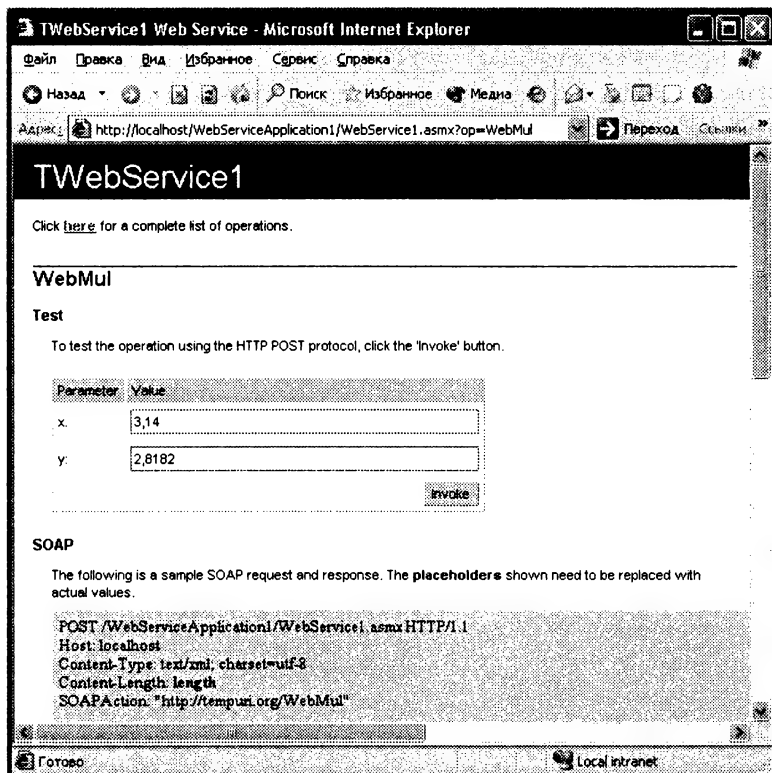


Рис. 6.17. Проверка Web-калькулятора

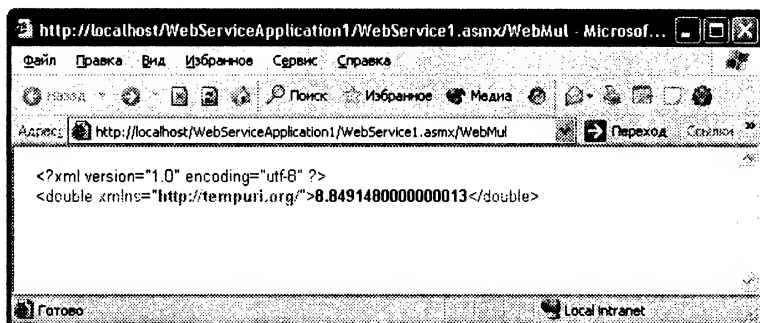


Рис. 6.18. Результат вычислений представлен в формате XML

- Введем произвольные значения, не забывая, что дробные числа в полях ввода надо указывать с запятой (если версия Windows русифицирована), и нажмем кнопку Invoke дистанционного вызова выбранного метода (рис. 6.17). Результат будет показан в браузере в наглядном формате XML (рис. 6.18).

6.3.4. Технология создания клиента Web-службы

Помимо создания собственно Web-служб система Delphi предоставляет, конечно, и средства создания клиентов, обращающихся к таким службам. Создаются подобные клиенты весьма просто. Сформируем для начала новое приложение (обычную форму Windows Forms или форму ASP.NET Web).

Пример: подготовка описания интерфейса Web-службы

В дальнейшем для обращения клиентских приложений к нашему (или любому другому) Web-сервису потребуется WSDL-файл, содержащий описание созданного интерфейса (в нем сейчас один метод умножения и метод HelloWorld, сгенерированный по умолчанию).

- Получить этот файл можно обращением:

```
http://localhost/WebsService1/WebService1.asmx?wsdl
```

- Можно щелкнуть на ссылке Service Description в окне браузера, где мы отлаживали список методов Web-службы (рис. 6.16). В результате в браузере появится XML-текст описания интерфейса, который надо сохранить как WSDL-файл (его расширение должно быть .WSDL).
- Дадим команду Project ➤ Add Web Reference (Проект ➤ Добавить Web-ссылку) — откроется диалоговое окно так называемого UDDI-браузера Borland. Он позволяет просматривать UDDI-каталоги, хранящие коллекции Web-служб, и выбирать подходящую дистанционную услугу. Мы сейчас укажем WSDL-файл, сформированный в предыдущем примере в ходе создания тестовой Web-службы (рис. 6.19).

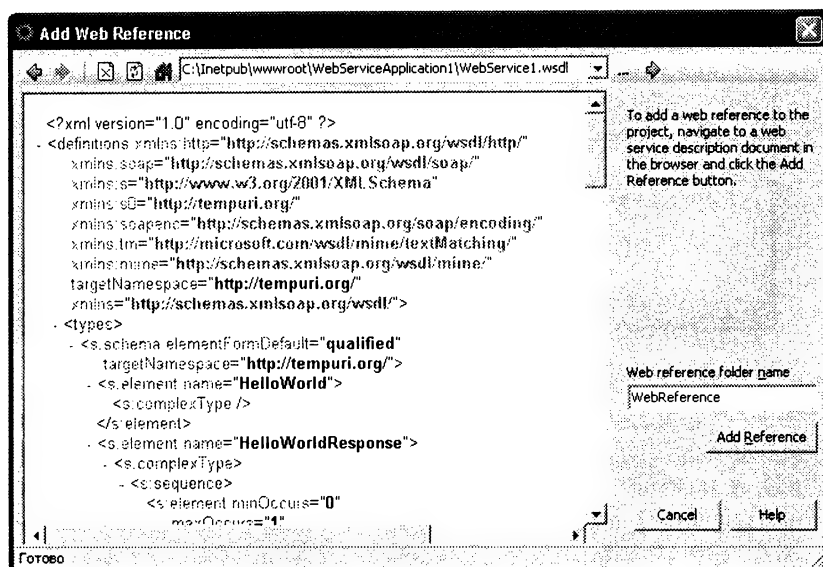


Рис. 6.19. Добавляем в проект описание Web-службы

Теперь нажмем кнопку Add Reference (Добавить ссылку), и описание выбранной нами Web-службы добавится к проекту. В частности, в папке проекта будет создан каталог WebReferences, а в нем разместится файл .WSDL.



С помощью команды Project ➤ Add Web Reference в один проект может быть добавлено множество разных Web-служб (прокси-классов). Эта команда введена специально для быстрого создания multifunctional приложений и систем, интегрирующих сторонние программы.

При работе с Web-службами надо точно знать, какой конкретно метод интерфейса Web-сервиса предназначен для вызова службы. В ходе вышеописанного процесса в проекте появился модуль WebReferences, который содержит описание класса (подобные классы нередко называются *прокси-классами*, *посредниками*), реализующего интерфейс, полученный в формате WSDL (рис. 6.21).

Просмотрев файл WebReference.WebService1.pas, мы увидим следующие строки.

type

```
[System.Diagnostics.DebuggerStepThroughAttribute]
[System.ComponentModel.DesignerCategoryAttribute('code')]
[System.Web.Services.WebServiceBindingAttribute
(Name= 'TWebService1Soap', Namespace='http://tempuri.org/')]
TWebService1 = class
```

```

(System.Web.Services.Protocols. SoapHttpClientProtocol)
    /// <remarks/>
    public
        constructor Create;
    /// <remarks/>
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute
    ('http://tempuri.org/HelloWorld',
    RequestNamespace='http://tempuri.org/',
    ResponseNamespace= 'http://tempuri.org/', Use= System.Web.
    Services.Description.SoapBindingUse.Literal, ParameterStyle=
    System.Web.Services.Protocols. SoapParameterStyle.Wrapped)]
        function HelloWorld: string;
    /// <remarks/>
        function BeginHelloWorld(callback: System.AsyncCallback;
    asyncState: System.Object): System.IAsyncResult;
    /// <remarks/>
        function EndHelloWorld(asyncResult: System.IAsyncResult):
    string;
    /// <remarks/>
    [System.Web.Services.Protocols.SoapDocumentMethodAttribute
    ('http://tempuri.org/WebMul',
    RequestNamespace='http://tempuri.org/', ResponseNamespace=
    'http://tempuri.org/',
    Use= System.Web.Services.Description.SoapBindingUse.
    Literal, ParameterStyle= System.Web.Services.Protocols.
    SoapParameterStyle.Wrapped)]

```

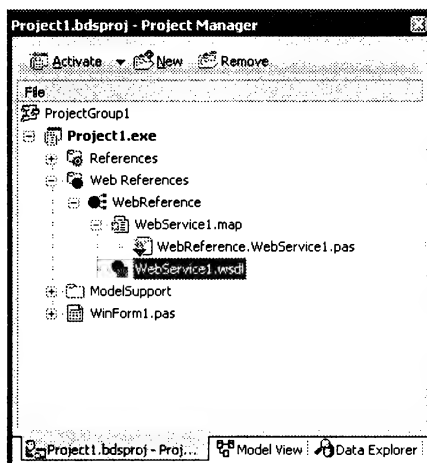


Рис. 6.21. Выбор описания Web-службы

```

function WebMul(x: System.Double; y: System.Double):
System.Double;
    /// <remarks/>
    function BeginWebMul(x: System.Double; y: System.Double;
callback: System.AsyncCallback;
    asyncState: System.Object): System.IAsyncResult;
    /// <remarks/>
    function EndWebMul(asyncResult: System.IAsyncResult):
System.Double;
end;

```

Видно, что здесь описан класс `TWebService1`, а в нем имеется метод `WebMul`.

Пример: организация доступа к Web-сервису из кода

Следующий шаг — программирование логики доступа к сервису и использование сервиса.

1. Допустим, мы хотим обращаться к услуге `WebMul` по нажатию на кнопку `Button1`. Для этого разместим на форме такую кнопку и создадим ее обработчик, а также разместим поле-надпись `label1`.
2. В обработчике сначала создадим объект Web-службы (экземпляр класса `TWebService1`), а затем обратимся к нужному нам методу `WebMul`.

```

procedure TWinForm1.Button1_Click(sender: System.Object;
e: System.EventArgs);
var
    ws: TWebService1;
    z: Double;
begin
    ws := TWebService1.Create; // создаем прокси-объект
    z := ws.WebMul( 3.14, 2.8182 ); // вызываем метод удаленной
Web-службы
    label1.Text := z.ToString; // выводим результат в текстовом
виде
end;

```

3. Чтобы данный код компилировался, надо включить в перечень доступных модулей ссылку на описание прокси-класса. Сделать это можно, например, командой `File ➤ Use Unit (Файл ➤ Использовать модуль)`, выбрав в диалоговом окне единственный модуль `WebReference.WebService1`.
4. Запустим программу, нажмем кнопку, и в поле надписи появится значение `8,849148`. Удобнее, конечно, разместить на форме два поля ввода, чтобы реализовать, например, дистанционный клиент-калькулятор.

Пример: программирование выдачи сложных значений

Возвращаемое Web-службой значение может иметь сколь угодно сложную структуру. При этом она будет автоматически переводиться из WSDL-описания в соответствующие типы данных Delphi.

1. Допустим, мы хотим, чтобы значение, рассчитанное в методе `WebMul`, представляло собой не одно число, а передавалось в виде следующей записи.

```
TWebAnswer=record
  Res: Double;
  ResStr: String;
end;
```

2. Пусть поле `Res` будет хранить численное значение произведения, а `ResStr` — его текстовое представление. Последнюю величину можно, конечно, получить гораздо более простыми способами, поэтому данный пример приведен исключительно для демонстрации интеллектуальных возможностей Delphi. Перепишем метод `WebMul` в новый вид с названием `WebAsk`:

type

```
TWebAnswer=record
  Res: Double;
  ResStr: String;
end;
```

```
TWebService1 = class(System.Web.Services.WebService)
{$REGION 'Designer Managed Code'}
strict private
  /// <summary>
  /// Required designer variable.
  /// </summary>
  components: IContainer;
  /// <summary>
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// </summary>
  procedure InitializeComponent;
{$ENDREGION}
strict protected
  /// <summary>
  /// Clean up any resources being used.
  /// </summary>
  procedure Dispose(disposing: boolean); override;
private
```

```

    { Private Declarations }
public
    constructor Create;

    // Sample Web Service Method
    [WebMethod]
    function HelloWorld: string;

    [WebMethod]
    function WebAsk(x,y: Double): TWebAnswer;

end;

```

Реализация его может быть такой.

```

function TWebService1.WebAsk(x,y: Double): TWebAnswer;
var n: TWebAnswer;
begin
    n.Res := x*y;
    n.ResStr := n.Res.ToString;
    Result := n;
end;

```

3. Выполним вышеописанную последовательность действий для Web-службы: соберем проект, получим WSDL-описание нашего интерфейса и сохраним его в файле, затем подготовим программу-клиент, добавим в нее Web-ссылку с помощью WDSL-файла и, наконец, переключимся в описание добавленного интерфейса на языке Delphi. В описании прокси-класса TWebService1 будут следующие строки.

```

    function WebAsk(x: System.Double; y: System.Double):
TWebAnswer;
    /// <remarks/>
    function BeginWebAsk(x: System.Double; y: System.Double;
callback: System.AsyncCallback;
        asyncState: System.Object): System.IAsyncResult;
    /// <remarks/>
    function EndWebAsk(asyncResult: System.IAsyncResult):
TWebAnswer;

```

4. Нужный нам метод WebAsk описан как возвращающий значение типа TWebAnswer. Просмотрев текущий файл, мы найдем его описание.

```

[System.Xml.Serialization.XmlTypeAttribute (Namespace=
'http://tempuri.org/')]
TWebAnswer = class
    ~ /// <remarks/>

```

```

public
  Res: System.Double;
  /// <remarks/>
  ResStr: string;
end;

```

Этот тип представлен не как структура, запись Паскаля (*record*), а как класс — это относится ко всем схемам обмена данными между Web-службами. Любые сложные структуры данных представляются в WSDL-описаниях в виде классов.

5. В классе TWebAnswer видны два наших поля — Res и ResStr. Как к нему обратиться? Так же, как и в предыдущем примере, созданием экземпляра нужного класса. А объект типа TWebAnswer будет сформирован автоматически.

```

var
  ws: TWebService1;
  z: TWebAnswer;
begin
  ws := TWebService1.Create;
  z := ws.WebAsk( 3.14, 2.8182 );
  label1.Text := z.ToString;
end;

```

Пример: обращение к реальным Web-службам

В заключение рассмотрим еще один пример. В Сети существует международная служба погоды Global Weather, показания которой поставляются, в частности, в виде Web-службы. Получить ее WSDL-описание можно обращением к такому Web-адресу.

```
http://www.webservices.com/globalweather.asmx?WSDL
```

1. Сохраним WSDL-файл на локальном диске, откроем пустое приложение, дадим команду Project ➤ Add Web Reference (Проект ➤ Добавить веб-ссылку) и выберем этот файл. Система Delphi сгенерирует новый файл проекта, в котором на языке Delphi будет храниться описание соответствующего прокси-класса.
2. Просмотрев файл, обнаружим нужный нам класс.

```

type
  GlobalWeather = class(System.Web.Services.Protocols.
    SoapHttpClientProtocol)
    /// <remarks/>
  public
    constructor Create;
    /// <remarks/>

```



```

[System.Web.Services.Protocols. SoapDocumentMethodAttribute
('http://www.webserviceX.NET/GetWeather', RequestNamespace=
'http://www.webserviceX.NET', ResponseNamespace='http://
www.webserviceX.NET', Use= System.Web.Services.Description.
SoapBindingUse.Literal,
ParameterStyle= System.Web.Services.Protocols.
SoapParameterStyle.Wrapped)]
function GetWeather(CityName: string; CountryName: string):
string;
/// <remarks/>
function BeginGetWeather(CityName: string; CountryName:
string; callback: System.AsyncCallback;
asyncState: System.Object): System.IAsyncResult;
/// <remarks/>
function EndGetWeather(asyncResult: System.IAsyncResult):
string;
/// <remarks/>
[System.Web.Services.Protocols. SoapDocumentMethodAttribute
('http://www.webserviceX.NET/GetCitiesByCountry',
RequestNamespace= 'http://www.webserviceX.NET', ResponseName
space='http://www.webserviceX.NET', Use=System.Web.Services.
Description. SoapBindingUse.Literal, ParameterStyle= System.
Web.Services.Protocols. SoapParameterStyle.Wrapped)]
function GetCitiesByCountry(CountryName: string): string;
/// <remarks/>
function BeginGetCitiesByCountry(CountryName: string;
callback: System.AsyncCallback;
asyncState: System.Object): System.IAsyncResult;
/// <remarks/>
function EndGetCitiesByCountry(asyncResult: System.
IAsyncResult): string;
end;
```

- По названиям функций можно предположить, что основным метод этой Web-службы — следующая функция.

```

function GetWeather(CityName: string; CountryName: string):
string;
```

Ее первый параметр — это название города, для которого сообщается погода, а второй — название страны. По всей видимости, названия эти должны быть указаны на английском языке. Кроме того, список доступных городов для определенной страны можно получить с помощью метода:

```

function GetCitiesByCountry(CountryName: string): string;
```

- Обращение к данной Web-службе формируется по известной схеме. Оно может выполняться, например, так:

```
var
    ws: GlobalWeather;
    s: String;
begin
    ws := GlobalWeather.Create;
    s := ws.GetWeather('Moscow', 'Russia');
    ...
```

Как показала проверка, такие параметры корректны, и Web-служба работает нормально. Правда, возвращаемое ею значение, видимо, для простоты, представляет собой не отдельный класс, как хотелось бы (например, с полями «температура», «влажность», «давление»), а одну непрерывную строку. Однако содержит она типичное XML-описание, которое можно разобрать «вручную», собственным программным кодом.



Разборку выражения можно упростить путем привлечения, например, стандартной библиотеки .NET Framework для разбора XML-документов, которая называется System.Xml.

Вот один из ответов этой Web-службы:

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
  <Location>Moscow / Vnukovo , Russia (UUWW) 55-39N 037-16E
</Location>
  <Time>Jul 27, 2005 - 08:30 AM EDT / 2005.07.27 1230 UTC
</Time>
  <Wind> from the W (270 degrees) at 9 MPH (8 KT):0</Wind>
  <Visibility> greater than 7 mile(s):0</Visibility>
  <SkyConditions> mostly cloudy</SkyConditions>
  <Temperature> 69 F (21 C)</Temperature>
  <DewPoint> 62 F (17 C)</DewPoint>
  <RelativeHumidity> 77%</RelativeHumidity>
  <Pressure> 29.85 in. Hg (1011 hPa)</Pressure>
  <Status>Success</Status>
</CurrentWeather>
```



Летняя дата (Delphi 2006 вышла в декабре 2005 года) объясняется участием автора книги в тестировании Borland Delphi и доступом к ранним версиям (field tests) продукта еще в июле.

Глава 7. Технология шаблонов (паттернов) проектирования

7.1. Что такое шаблоны проектирования

Концепция шаблонов¹ проектирования развивает один из основных принципов объектно-ориентированного программирования — повторное использование кода. Вместо того чтобы каждый раз заново проектировать схожие программные архитектуры, можно воспользоваться готовыми строительными элементами. Точно так же, как приложение складывается из компонентов: кнопок, меню, модулей связи с базами данных и ресурсами Интернета — модели тоже содержат немалое число часто повторяющихся структур. Эти структуры были выделены специалистами по программной инженерии и повторно используемые описания.

Каждый шаблон — это архитектурная концепция, элемент модели будущего приложения. Он был получен не столько в результате точной аналитической работы, сколько путем обобщения многолетнего опыта экспертов в области проектирования крупных систем. Таких паттернов было выделено несколько десятков. Шаблон фактически представляет собой маленькое решение некоторой типовой и часто встречающейся подзадачи проектирования программной системы. При этом структура шаблона проектирования такова, что он легко настраивается на конкретную прикладную модель.

1 Наряду с определением «шаблон» в российской компьютерной литературе прижился его англоязычный аналог «паттерн» (от англ. pattern). В частности, переведенная на русский язык книга основоположников этого направления называлась «Паттерны проектирования» («Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001 г.). В первом издании она была выпущена экспертами корпорации IBM Эриком Гамма, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом. Эти специалисты по проектированию сложных программных архитектур выбрали для своего неформального объединения название «Gang of Four» (банда четырех) — в честь одноименной рок-группы, которая исполняла в 70-х годах столь прогрессивную музыку, что так и осталась непонятой. Мы обращаем внимание на это обстоятельство только потому, что в Delphi классический набор шаблонов назван GoF (сокращение от Gang of Four).

В шаблоне можно выделить четыре основные составляющие.

- **Имя:** стандартное, предложенное автором название, отражающее предназначение шаблона. Изменению не подлежит.
- **Задача, решаемая шаблоном.** Обычно в дополнение к абстрактному описанию задачи указываются характерные элементы конкретного контекста использования шаблона и условия, при которых шаблон может эффективно действовать.
- **Способ решения задачи.** Представлен набором классов и методов, которые предназначены для ее решения.
- **Результат применения.** Описан в виде последствий и возможных ограничений, которые данный шаблон вносит в проектную архитектуру.

Такое описание шаблона довольно абстрактно, и по нему трудно понять, чем шаблон отличается, например, от стандартного класса библиотеки Delphi, реализующего функциональность коллекции. А разница в том, что такой класс всего лишь предоставляет разработчику удобные средства обработки набора объектов, однако сам по себе он не содержит никаких подсказок и свойств, которые можно было бы применить на достаточно высоком уровне абстрагирования (абстрагирования от понятия свойств и методов) как для решения конкретной задачи, так и для включения в общую схему программы. Шаблон объединяет не только классы типовых структур, но и взаимосвязи между ними, и методы их использования и обработки, представляя, таким образом, фактически метакласс — не просто готовый класс, который подчас неизвестно как использовать, а целостную модель некоторого процесса решения, порождающую семейство связанных классов, настраиваемую на контекст конкретной области применения.

7.2. Группы шаблонов

Все задействованные в Delphi шаблоны, согласно классификации GoF, делятся на три большие группы: *поведенческие шаблоны (Behavioral)*, *порождающие шаблоны (Creational)* и *структурные шаблоны (Structural)*.

Шаблоны поведения, как это явствует из их названия, охватывают задачи управления системой и взаимосвязями между классами.

Порождающие шаблоны ответственны за создание прикладных объектов программы и их взаимодействие. Это достигается скрытием прикладных классов внутри абстрактных классов шаблонов, а взаимодействие объектов в системе организуется исключительно на уровне интерфейсов новых классов, единым универсальным способом.

Структурные шаблоны упрощают формирование крупных структур из более мелких классов.

Создается любой шаблон выбором на панели инструментов пункта *Node by Pattern* и последующего указания в диалоговом окне Мастера шаблонов названия нужного шаблона. В результате в Дизайнере модели появится набор всех нужных классов шаблона с настроенными взаимосвязями (рис. 7.1).

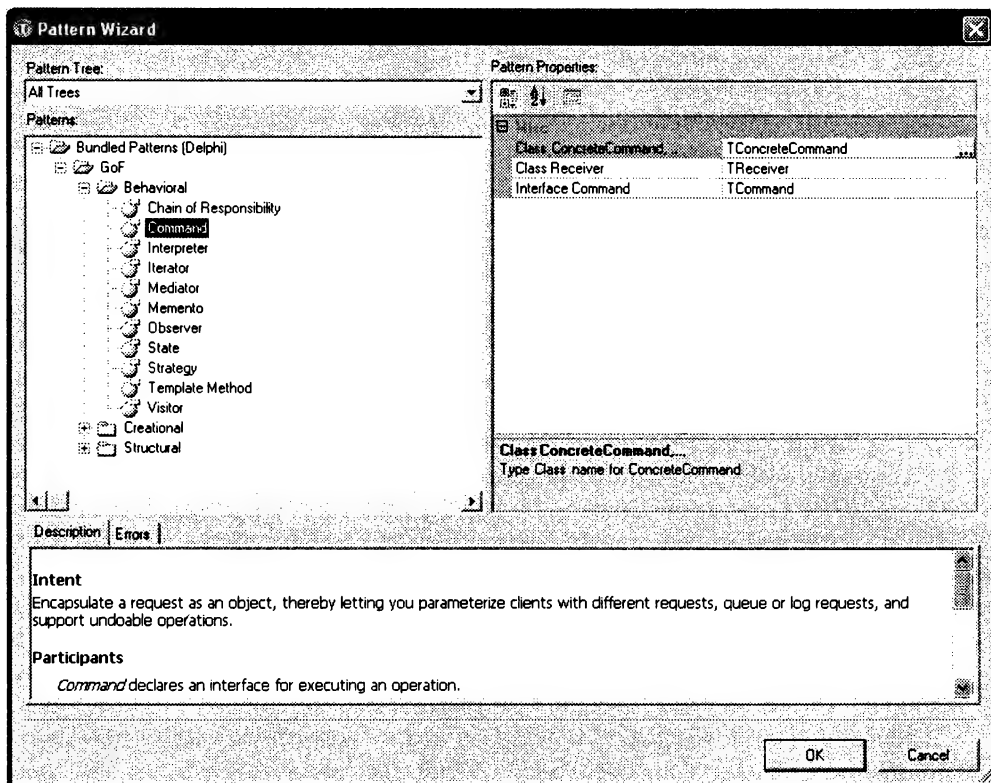


Рис. 7.1. Окно Мастера шаблонов



Создание шаблона возможно в пространстве моделирования Delphi. Чтобы в него попасть, надо в Менеджере проекта переключиться на вкладку **Model View** (В виде модели) или вызвать явно Просмотрщик модели командой главного меню **View > Model View** (Вид > В виде модели). После этого надо выбрать один из модулей модели и дважды на нем щелкнуть — в центральном окне Delphi появится вкладка **Diagram** с диаграммами текущей модели.

7.3. Порождающие шаблоны

7.3.1. Шаблон **Abstract Factory** (Абстрактная фабрика)

Этот шаблон предоставляет универсальный интерфейс взаимодействия объектов, каждый из которых может принадлежать произвольным классам проекта, скрытым от разработчика. Это позволяет отделить функциональность системы от внутренней реализации каждого класса. Обращение к ним становится возможным через абстрактные интерфейсы.

```
IAbstractProduct = interface
    function Func1: integer;
end;
```

В реализующем его классе `TConcreteProduct`, соответственно, надо запрограммировать метод `Func1`.

```
TConcreteProduct = class(TObject, IAbstractProduct)
    function Func1: integer;
end;

function TConcreteProduct.Func1: integer;
begin
    Result := 5;
end;
```

Создание данного продукта в коде уже было сформировано автоматически в момент добавления шаблона к модели.

```
function TConcreteFactory.CreateIAbstractProduct:
IAbstractProduct;
begin
    Result := TConcreteProduct.Create();
end;
```

Теперь добавим к шаблону новый конкретный продукт — выберем объект `AbstractFactory` на диаграмме (образ шаблона в овале) и в его контекстном меню дадим команду `Add > Concrete Product` (Добавить > Конкретный продукт). К модели добавится новый класс, который мы назовем `TConcreteProduct2`. Он автоматически наследует интерфейс `IAbstractProduct`, поэтому в нем также надо указать реализацию метода `Func1`.

```
TConcreteProduct2 = class(TObject, IAbstractProduct)
    public
        function Func1: integer;
end;

function TConcreteProduct2.Func1: integer;
begin
    Result := 10;
end;
```

Кроме того, требуется вручную (начиная с Дизайнера модели) добавить в Абстрактную фабрику метод создания нового продукта (обратите внимание, что каждая функция возвращает один и тот же абстрактный интерфейс).

```
IAbstractFactory = interface
    function CreateIAbstractProduct: IAbstractProduct;
    function CreateIAbstractProduct2: IAbstractProduct;
end;
```

И в реальную фабрику — также, только уже с реализацией метода создания нового продукта.

```

TConcreteFactory = class(TObject, IAbstractFactory)
public
    function CreateIAbstractProduct: IAbstractProduct;
    function CreateIAbstractProduct2: IAbstractProduct;
end;

function TConcreteFactory.CreateIAbstractProduct2:
IAbstractProduct;
begin
    Result := TConcreteProduct2.Create();
end;

```

На этом реализация Абстрактной фабрики завершена, и можно перейти к ее использованию. В обработчике нажатия на кнопку надо описать две переменные интерфейсов.

```

var cpI: IAbstractProduct;
    afI: IAbstractFactory;

```

С их помощью мы и будем организовывать доступ к ресурсам фабрики. В коде самого обработчика укажем команду создания реальной фабрики, реализующей интерфейс `IAbstractFactory`

```
afI := TConcreteFactory.Create;
```

и команду создания с помощью фабрики конкретного продукта

```
cpI := afI.CreateIAbstractProduct2;
```

Теперь через интерфейс `IAbstractProduct` мы можем обращаться к конкретному экземпляру любого из классов `TConcreteProduct` и `TConcreteProduct2`.

```

procedure TWinForm.Button1_Click(sender: System.Object;
e: System.EventArgs);
var cpI: IAbstractProduct;
    afI: IAbstractFactory;
begin
    afI := TConcreteFactory.Create;
    cpI := afI.CreateIAbstractProduct;

    label1.Text := cpI.Func1.ToString;

end;

```

В данном случае в поле `label1` будет показано значение 5, а в следующем случае:

```

procedure TWinForm.Button1_Click(sender: System.Object; e:
System.EventArgs);
var cpI: IAbstractProduct;

```



```

    afI: IAbstractFactory;
begin
    afI := TConcreteFactory.Create;
    cpI := afI.CreateIAbstractProduct2; // создание экземпляра
TConcreteProduct2

    label1.Text := cpI.Func1.ToString;

end;

```

будет выведено значение 10.

7.3.2. Шаблон Builder (Строитель)

Данный шаблон полезен, когда требуется конструировать объекты сложной структуры, причем процесс их конструирования может состоять из нескольких скрытых (отделенных от пользователя) шагов. Более того, последовательность и содержание этих шагов могут меняться в зависимости от исходных требований и параметров Строителя.

Работает данный шаблон так. Создается экземпляр Конкретного строителя `TConcreteBuilder`, далее — связанный с ним экземпляр Распорядителя `TDirector`, ответственного за формирование итогового продукта. В дальнейшем Распорядитель управляет работой Конкретного строителя, отдавая ему команды на создание той или иной части финального продукта. Команды эти обычно скрыты внутри единого метода `Construct` Распорядителя.



Таких методов может быть и несколько — каждый из них определяет собственный способ создания итогового продукта, доступ к которому, тем не менее, всегда происходит через универсальный абстрактный интерфейс.

Допустим, у нас имеется класс `MyProduct` со строковым полем `field`, содержимое которого формируется последовательностью некоторых операций (например, обращением к базе данных и постепенным наращиванием строки символ за символом), причем эта последовательность зависит от конкретной реализации.

Добавим к проекту новый шаблон `Builder` (рис. 7.3).

Введем на диаграмму новый класс `MyProduct` с полем `field`, после чего свяжем этот класс с интерфейсом `IProduct` продукта Строителя.

```

MyProduct = class (TObject, IProduct)
public var
    field: string;
end;

```

Интерфейс Строителя расширим двумя методами `BuildPart1` и `BuildPart2`, которые станут последовательно создавать объект-продукт и расширять его поле `field` какими-то значениями.

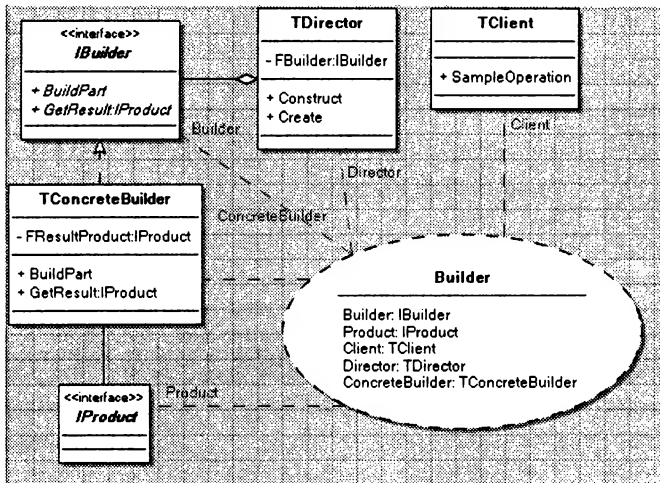


Рис. 7.3. Шаблон Строитель в Дизайнере модели

```

IBUILDER = interface
    function GetResult: IProduct;
    procedure BuildPart1;
    procedure BuildPart2;
end;

```

```

IProduct = interface
end;

```

Эти методы реализуем в классе Конкретного строителя.

```

TConcreteBuilder = class(TObject, IBUILDER)
    strict private var
        FResultProduct: IProduct;

    public
        procedure BuildPart1;
        procedure BuildPart2;

        ///<issubroutine>True</issubroutine>
        function GetResult: IProduct;
end;

```

```

// первый этап
procedure TConcreteBuilder.BuildPart1;
begin

```

```

// создаем продукт

```

```

    FResultProduct := MyProduct.Create;

    // начинаем формировать его содержимое:
    MyProduct(FResultProduct).field := 'a';
end;

// второй этап
procedure TConcreteBuilder.BuildPart2;
begin
    // продолжаем наращивать содержимое
    MyProduct(FResultProduct).field :=
        MyProduct(FResultProduct).field + 'b';
end;

// возвращаем продукт
function TConcreteBuilder.GetResult: IProduct;
begin
    Result := FResultProduct;
end;

```

Класс Распорядителя составлен из двух методов: конструктора, в котором он получает интерфейс Строителя, и процедуры конструирования итогового продукта.

```

TDirector = class
    strict private var
        ///<link>aggregation</link>
        FBuilder: IBuilder;

    public
        ///<issubroutine>True</issubroutine>
        procedure Construct;
        constructor Create(ABuilder : IBuilder);
end;

```

Вот как может быть реализована процедура Construct — в ней последовательно вызываются два этапа формирования продукта.

```

procedure TDirector.Construct;
begin
    FBuilder.BuildPart1();
    FBuilder.BuildPart2();
end;

```

Теперь к данным классам можно обращаться из прикладного кода, например, из обработчика нажатия на кнопку.

```

procedure TwinForm.Button1_Click(sender: System.Object; e:
System.EventArgs);
var
    Builder :TConcreteBuilder;
    Product :IProduct;
begin
    // создаем Конкретный строитель
    Builder := TConcreteBuilder.Create();

    // создаем Распорядитель, использующий Строителя
    // и сразу вызываем метод конструирования.
    // Таким образом удастся обойтись без
    // введения локальной переменной для Распорядителя
    TDirector.Create(Builder).Construct();

    // получаем объект-продукт
    Product := Builder.GetResult();

    // результатом будет строка "ab"
    labell1.Text := MyProduct(Product).field;
end;

```

7.3.3. Шаблон Factory Method (Фабричный метод)

Фабричные методы развивают концепцию Абстрактной фабрики. Они позволяют создавать объекты самых разных классов, доступ к которым осуществляется через единый универсальный продуктовый интерфейс (рис. 7.4).

Интерфейс `IProduct` используется в прикладной программе для доступа к стандартному набору возможностей объектов, которые реально, вполне воз-

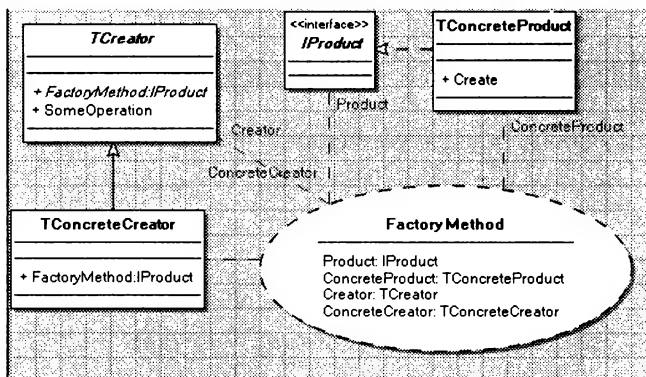


Рис. 7.4. Шаблон Фабричный метод в Дизайнере модели

можно, представляют собой экземпляры самых разных классов. Допустим, этот интерфейс (он формируется автоматически добавлением к модели текущего шаблона) состоит из двух методов: устанавливающего значение некоторого поля и возвращающего это значение.

```
IProduct = interface
  procedure SetFunc(d:integer);
  function GetFunc: integer;
end;
```

Пусть у нас имеется два разных продуктовых класса: TConcreteProduct и TConcreteProduct2. У одного из них имеется поле field1, а у другого — field2.

```
TConcreteProduct = class(TObject, IProduct)
  public
  var
    field:integer;

  constructor Create;

  procedure SetFunc(d:integer);
  function GetFunc: integer;
end;

TConcreteProduct2 = class(TObject, IProduct)
  public var
    field2:integer;

  constructor Create;
  procedure SetFunc(d:integer);
  function GetFunc: integer;
end;
```

Обратите внимание, что внутренние структуры этих классов несколько различаются.

Их методы реализуем схожим образом.

```
function TConcreteProduct.GetFunc: integer;
begin
  Result := field
end;

function TConcreteProduct2.GetFunc: integer;
begin
  Result := field2
end;
```

```

procedure TConcreteProduct.SetFunc(d: integer);
begin
    field := d;
end;

procedure TConcreteProduct2.SetFunc(d: integer);
begin
    field2 := d;
end;

constructor TConcreteProduct.Create;
begin
    inherited;

    field := 100;
end;

constructor TConcreteProduct2.Create;
begin
    inherited;
    field2 := 200;
end;

```

Нам требуется обеспечить взаимодействие с экземплярами этих классов через единый интерфейс `IProduct`. Для этого в шаблоне имеется абстрактный класс Создатель.

```

TCreator = class abstract
    public
        ///<issubroutine>True</issubroutine>
        procedure SomeOperation;
        function FactoryMethod: IProduct; virtual; abstract;
    end;

```

В нем нас интересует фабричный метод `FactoryMethod`, создающий нужный объект-продукт.

Как определить, в каком случае какой объект создавать? Для этого либо создается несколько конкретных классов-наследников класса `TCreator` (если вариантов продуктов немного), либо один такой класс, в который в качестве параметра передается запрос на создание экземпляра нужного класса.

В нашем случае можно обойтись одним Конкретным создателем, расширив его фабричный метод параметром `id` — на его основе и будет создаваться нужный экземпляр, интерфейс которого возвращается главной программе.

```

function TConcreteCreator.FactoryMethod(id: integer): IProduct;
begin

```

```

case id of
    1:Result := TConcreteProduct.Create();
    2:Result := TConcreteProduct2.Create();
end;

end;

```

Как правило, действия над объектами, доступ к которым возможен через универсальный интерфейс, также по большей части скрываются и выделяются в методы Создателя, например, так.

```

function TCreator.SomeOperation: Integer;
var
    Product1, Product2 :IProduct;
begin
    // ...
    Product1 := FactoryMethod(1);
    Product2 := FactoryMethod(2);
    Result := Product1.GetFunc + Product2.GetFunc;
    // ...
end;

```

Обращение к данному методу из прикладной формы при нажатии на кнопку может быть, например, таким.

```

procedure TWinForm.Button1_Click(sender: System.Object; e:
System.EventArgs);
    var IC: TConcreteCreator;
begin
    IC := TConcreteCreator.Create();
    label1.Text := IC.SomeOperation.ToString
end;

```

В поле label1 будет показано число 300. При этом прикладной разработчик не будет знать, что происходит внутри метода SomeOperation.

7.3.4. Шаблон Prototype (Прототип)

В ряде проектов возникает необходимость создания множества объектов — экземпляров не очень большого числа классов, каждый из которых требует предварительной и довольно объемной настройки. Такую настройку можно выполнять вручную, а можно заранее подготовить набор объектов-прототипов — по одному на каждый класс — и при необходимости просто создавать их копии-клоны (рис. 7.5).

Данный шаблон решает эту проблему, предоставляя единый интерфейс клонирования: IPrototype.

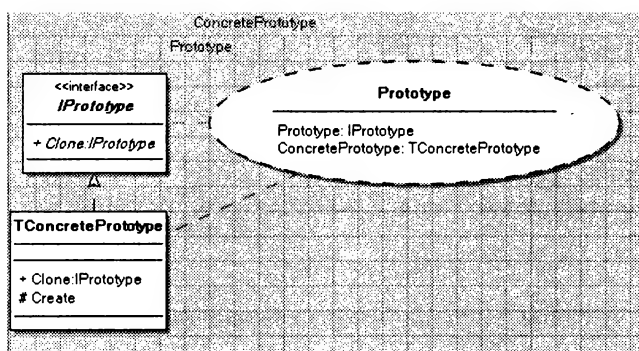


Рис. 7.5. Шаблон Прототип в Дизайнере модели

type

```

IPrototype = interface
    function Clone: IPrototype;
end;

```

В соответствии с шаблоном прикладной программе достаточно обратиться к методу Clone любого объекта, чтобы тут же получить его настроенную копию со всеми настроенными (текущими) значениями полей.

Допустим, в программе имеются два класса: TConcretePrototype и TConcretePrototype2, реализующие этот интерфейс. Структуры их в общем случае могут сильно различаться (так, в нашем случае имеются два поля с одинаковыми названиями field, но содержащие данные разных типов: целое и строка), а при создании вполне может выполняться объемная настройка внутренних свойств.

```

TConcretePrototype = class(TObject, IPrototype)
    strict protected
        constructor Create(APrototype :TConcretePrototype);
overload;

    public
        constructor Create;overload;

    var
        field:string;

        function Clone: IPrototype;
end;

TConcretePrototype2 = class(TObject, IPrototype)
    public
        constructor Create;overload;

```



```

var
    field:integer;

    function Clone: IPrototype;

    strict protected
        constructor Create(APrototype :TConcretePrototype2);
overload;
    end;

```

В конструкторах выполняется некая дополнительная настройка свойств.

```

    constructor TConcretePrototype.Create (APrototype :
TConcretePrototype);
    begin
        inherited Create;
        field := 'CP';
    end;

    constructor TConcretePrototype2.Create (APrototype :
TConcretePrototype2);
    begin
        inherited Create;
        field := 50;
    end;

```

Клонирование осуществляется в конструкторе, которому передается копия самого вызывающего объекта.

```

    function TConcretePrototype.Clone: IPrototype;
    begin
        Result := TConcretePrototype.Create(self);
    end;

    function TConcretePrototype2.Clone: IPrototype;
    begin
        Result := TConcretePrototype2.Create(self);
    end;

```

Теперь на базе этих классов в прикладной программе надо создать первые экземпляры, которые и будут впоследствии клонироваться.

```

var IC1, IC2: IPrototype;

...

// создаем по одному прототипу каждого класса:
IC1 := TConcretePrototype.Create();
IC2 := TConcretePrototype2.Create();

```

```
// теперь на их основе можно строить произвольные схемы
клонирования:
```

```
label1.Text := TConcretePrototype(IC1.Clone).field.ToString
             + TConcretePrototype2(IC2.Clone).field.ToString;
```

Обращением к методу `Clone` мы формируем копию нужного экземпляра класса и затем сразу передаем ее в качестве конструктора. Работа при этом ведется на уровне интерфейса `IPrototype`.

7.3.5. Шаблон Singleton (Одиночка)

Существует немало случаев, когда некоторый класс в программе должен быть представлен единственным экземпляром. Например, в библиотеке Delphi VCL имеется глобальная переменная `Application` (экземпляр класса `TApplication`), предоставляющая доступ к системным характеристикам самой программы в процессе выполнения. Очевидно, что экземпляр этого класса может быть только один, а доступ к нему предоставляет глобальная переменная `Application`. Однако теоретически ничто не мешает разработчику создать новый экземпляр класса `TApplication` (рис. 7.6).

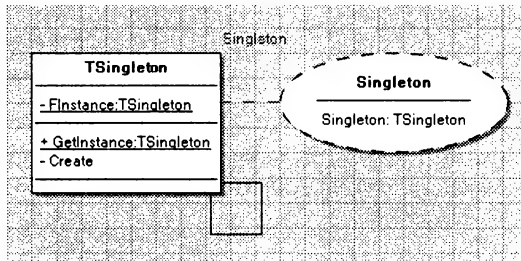


Рис. 7.6. Шаблон Одиночка в Дизайнере модели

Шаблон Одиночка позволяет создать только один экземпляр класса, а доступ к нему осуществляется через указатель — внутреннее поле `FInstance`, к которому можно обращаться через метод `GetInstance`.

```
type
  TSingleton = class
    strict private
      class var
        FInstance: TSingleton;

    constructor Create;

  public
    class function GetInstance: TSingleton;
  end;
```

Реализация этого метода прозрачна — в ней обеспечивается постоянная работа с единственным, причем одним и тем же экземпляром данного класса:

```
class function TSingleton.GetInstance: TSingleton;
begin
    If FInstance = nil Then
    begin
        FInstance := TSingleton.Create();
    end;
    Result := FInstance;
end;
```

При этом программист всегда использует только данный метод, не разбираясь, создавался ли предварительно данный экземпляр явно — эта проверка происходит автоматически.



Отметим также, что конструктор скрыт в private-разделе класса TSingleton, чтобы к нему нельзя было получить доступ, а переменная FInstance описана как переменная класса, что допускает абстрактные операции над классом без явного создания объекта. Такая тактика хорошо подходит для реализации данной задачи.

Часто конкретные классы, в которых должна быть реализована функциональность Одиночки, формируются непосредственно в рамках класса TSingleton. Добавим, например, ему новое поле field: integer, после чего организуем работу следующим образом.

```
var sg: TSingleton;

...

// первое обращение - экземпляр класса создается
автоматически
sg := TSingleton.GetInstance;

sg.field := 5;
labell.Text := sg.field.ToString;

(* следующее обращение снова выполняется с помощью
GetInstance - но работа происходит с одним и тем же объектом!
*)

sg := TSingleton.GetInstance;

// будет показано число "5":
labell.Text := sg.field.ToString;
```

7.4. Структурные шаблоны

Структурные шаблоны упрощают процесс формирования сложных структур из имеющихся классов.

7.4.1. Шаблон Adapter (Адаптер)

Данный шаблон позволяет состыковать два интерфейса. Это требуется, когда заказчику надо взаимодействовать с объектами системы через свой интерфейс, но объекты внутри этой системы предоставляют другие интерфейсы, не совместимые с клиентским (рис. 7.7).

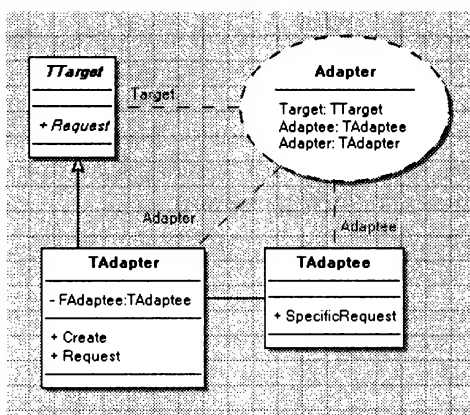


Рис. 7.7. Шаблон Адаптер в Дизайнере модели

TTarget — это интерфейс, который задан с внешней стороны по отношению к имеющейся системе. Пусть в нем имеется метод GetCodeString, который должен возвращать некоторую строку. А в адаптируемом к нему интерфейсе TAdaptee вообще нет никаких методов, а только числовое поле field. Функцию адаптера выполняет класс TAdapter:

```

type
  TTarget = class abstract
  public
    ///<issubroutine>True</issubroutine>
    function GetCodeString: string; virtual; abstract;
  end;

  TAdaptee = class
  public
    var
      field: integer;
  end;

```

```

end;

TAdapter = class(TTarget)
  strict private var
    FAdaptee:TAdaptee;

  public
    ///<issubroutine>True</issubroutine>
    function GetCodeString: string; override;
    constructor Create(AAdaptMe :TAdaptee);
end;

```

Адаптер реализует переходные функции путем наследования базового интерфейса TTarget, за счет чего внешний пользователь обращается к адаптеру как объекту, реализующему абстрактный интерфейс TTarget. А для взаимодействия с «местным» адаптируемым объектом внутри адаптера имеется поле FAdaptee, которое хранит этот адаптируемый объект (он передается в адаптер сразу в момент конструирования).

```

constructor TAdapter.Create(AAdaptMe :TAdaptee);
begin
  inherited Create;
  FAdaptee := AAdaptMe;
end;

```

Преобразование адаптируемого интерфейса к целевому реализуется внутри метода GetCodeString, унаследованного от интерфейса TTarget. Например, так.

```

function TAdapter.GetCodeString: string;
begin
  Result := FAdaptee.field.ToString;
end;

```

Пусть в прикладной программе доступен адаптируемый объект:

```
var ade: TAdaptee;
```

и подготовлен объект-адаптер

```
var ad: TAdapter;
```

Тогда, после того как объект с адаптируемым интерфейсом создан и настроен:

```

ade:= TAdaptee.Create;
ade.field := 5;

```

формируется объект-адаптер

```
ad:= TAdapter.Create(ade);
```

В нем оказывается экземпляр `ade`, а внешне доступен интерфейс `TTarget`:

```
label1.Text := ad.GetCodeString;
```

7.4.2. Шаблон Bridge (Мост)

Шаблон Мост отделяет интерфейсную часть от ее реализации, причем не так, как это выполнено в абстрактных классах, где модификация их описания вызывает необходимость внесения изменений во все классы-наследники, а независимо. То есть, изменив описание интерфейса с помощью Моста, можно не выполнять массовой переделки всех его подклассов (рис. 7.8).

Внешний интерфейс в этом шаблоне представлен абстрактным классом `TAbstraction`.

```
TAbstraction = class abstract
    ///<link>aggregation</link>
    FImpl:TImplementor;

    strict protected
        function getImplementor: TImplementor;
        procedure setImplementor(im: TImplementor);

    public
        ///<issubroutine>True</issubroutine>
        function SomeOperation: Integer ;
end;
```

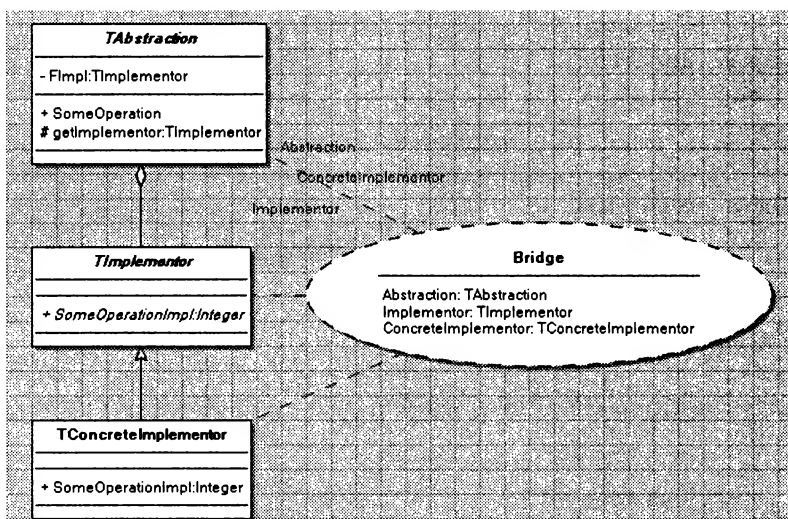


Рис. 7.8. Шаблон Мост в Дизайнере модели

```

procedure TAbstraction.setImplementor(im: TImplementor);
begin
    FImpl := im;
end;

```

На его основе формируется конкретный интерфейс-потомок, который и используется в проекте, например класс `MyClass`:

```

MyClass = class(TAbstraction)
end;

```

Однако метод `SomeOperation` в класс `MyClass` включать не обязательно. Он в данном случае не требуется для реализации нужной функциональности, как в случае с обычным наследованием. В этом методе обычно находится только обращение к нужному методу класса реализации, поэтому в классе-наследнике `MyClass` этот метод можно не реализовывать. А на уровне `TAbstraction` в методе происходит обращение к полю `FImpl` — экземпляру класса `TImplementor`, который и занимается реализацией интерфейса `TAbstraction` (она полностью скрыта от внешнего пользователя класса `TAbstraction`).

```

function TAbstraction.SomeOperation: Integer ;
begin
    result := getImplementor().SomeOperationImpl(5);
end;

```

За реализацию отвечает абстрактный класс `TImplementor`

```

TImplementor = class abstract
    public
        function SomeOperationImpl(i:Integer): Integer;
virtual; abstract;
end;

```

Скрытая ссылка на него хранится в классе описания интерфейса. По аналогии с классом `MyClass`, реализация данного класса также формируется на основе наследования.

```

TConcreteImplementor = class(TImplementor)
    public
        function SomeOperationImpl(i:Integer): Integer; override;
end;

function TConcreteImplementor.SomeOperationImpl (i:Integer):
Integer;
begin
    result := i*i;
end;

```

Кроме того, класс `MyClass` надо дополнить конструктором, в котором создается экземпляр класса реализации.

```
MyClass = class(TAbstraction)
    public
        constructor Create;
end;

constructor MyClass.Create;
begin
    inherited;
    setImplementor( TConcreteImplementor.Create );
end;
```



Обычно класс `TConcreteImplementor` не используется напрямую. Его возможности наследуются несколькими классами-потомками, что позволяет очень гибко менять реализацию одного и того же интерфейса `TAbsraction`. Для этого, например, достаточно в классах-наследниках `TAbsraction` лишь изменить конструктор, указав тот или иной класс реализации.

Тогда в прикладном коде обращение к этому классу может записываться так.

```
var ad: TAbstraction;
...
ad := MyClass.Create;

label1.Text := ad.SomeOperation.ToString; // обращение к
методу SomeOperation интерфейса
```

7.4.3. Шаблон Composite (Компоновщик)

Шаблон Компоновщик предоставляет интерфейс, обеспечивающий одинаковую работу как с отдельным классом, так и с набором классов, поддерживающим те же методы. Это полезно в случаях, когда команду надо одновременно выполнить большому числу схожих по структуре объектов, хотя и принадлежащих, возможно, разным классам (рис. 7.9).

Допустим, имеется класс `TLeaf`, аналогов которого в прикладной программе (как и их экземпляров) может быть множество.

```
TLeaf = class(TComponent)
    public
        procedure SampleOperation(i :Integer);override;
end;
```

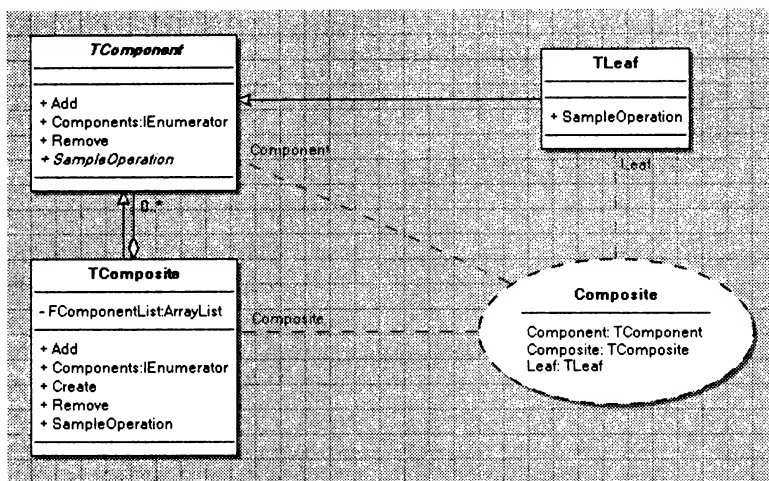



Рис. 7.9. Шаблон Компоновщик в Дизайнере модели

Работа с такими базовыми классами обеспечивается абстрактным классом TComponent

```
TComponent = class abstract
  public
    function Components: IEnumerator; virtual;
    procedure Remove(AComponent : TComponent); virtual;
    procedure Add(AComponent : TComponent); virtual;
    procedure SampleOperation(i : Integer); virtual; abstract;
end;
```

Класс TComponent позволяет добавлять и удалять отдельные объекты, предоставляет универсальный интерфейс IEnumerator к набору элементов, а также обеспечивает выполнение метода SampleOperation для каждого из них. При этом любой из элементов данного класса, в свою очередь, может быть представителем как отдельного объекта TLeaf, так и составного — экземпляра TComponent. Таким образом можно строить древовидные структуры отношений между классами, а обращение к методу SampleOperation вызывает автоматический рекурсивный обход этого дерева.

Отметим возможную реализацию прикладного метода SampleOperation, который предназначен для вызова из прикладной программы. В нем задействован новый оператор Delphi **for ... in ...**, предназначенный для перебора всех элементов коллекции.

```
procedure TComposite.SampleOperation(i : Integer);
var current: TComponent;
begin
  for current in FComponentList do
```

```
begin
current.SampleOperation(i);
end
end;
```

Прикладное построение дерева может быть, например, таким.

```
var ad: TComponent; // главный интерфейс
    lf: TLeaf; // простой узел
    c1, c2: TComposite; // составные узлы
...

ad:= TComposite.Create;

// добавляем простой узел
lf:= TLeaf.Create;
ad.Add(lf);

// создаем будущий составной узел
c1:= TComposite.Create;

// строим его содержимое из двух простых:
c1.Add(lf); c1.Add(lf);

// добавляем составной узел в главный интерфейс:
ad.Add(c1);

// добавляем еще один простой узел:
ad.Add(lf);

// выполняем наш метод SampleOperation(), который будет
автоматически вызван для каждого простого узла, связанного с
объектом ad:
ad.SampleOperation(3);
```

7.4.4. Шаблон Decorator (Декоратор)

Декоратор, как и другие шаблоны данной группы, предоставляет возможность гибкой модификации интерфейса, предназначенного для конечного разработчику. В частности, Декоратор позволяет расширять некоторый класс новыми функциями в случаях, когда наследование этого класса невозможно или приводит к усложнению проекта (если, например, приходится расширять интерфейс на протяжении всей разработки, и каждый раз реализовывать дополнение во множестве классов-наследников сложно; или когда требуется модифицировать поведение объекта динамически, во время работы программы), — рис. 7.10.

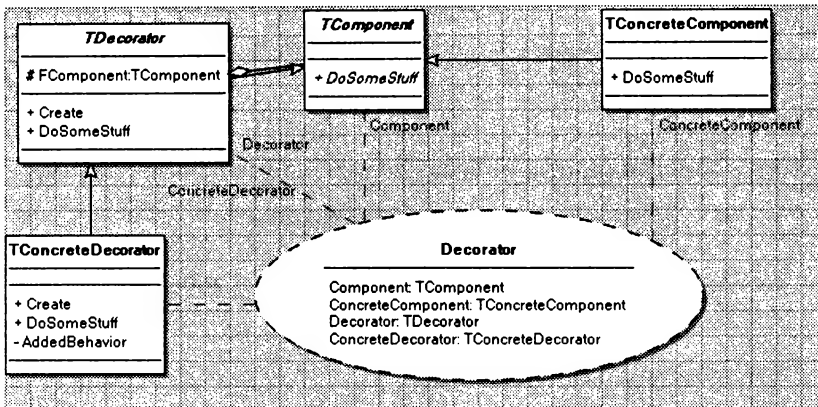


Рис. 7.10. Шаблон Декоратор в Дизайнере модели

Шаблон Декоратор предлагает основной интерфейс `TComponent` — его расширенная функциональность реализуется в классе `TConcreteComponent`. Физически же формируются наследники Декоратора, которые, с одной стороны, хранят ссылку на экземпляр класса, реализующего интерфейс `TComponent`, чтобы он выполнил основной вызванный метод, а с другой стороны, дополняют ее собственными действиями.

Допустим, имеется следующее описание класса.

```

TComponent = class abstract
public
    procedure DoSomeStuff; virtual; abstract;
end;

```

Он реализован следующим образом.

```

TConcreteComponent = class(TComponent)
public
    ///<issubroutine>True</issubroutine>
    procedure DoSomeStuff; override;
end;

```

Допустим, мы желаем расширить функциональность метода `DoSomeStuff` с помощью Декоратора. Вот как записывается декорирующий класс.

```

TDecorator = class abstract(TComponent)
strict protected var
    FComponent: TComponent; // ссылка на оригинальный
    компонент

public
    procedure DoSomeStuff; override;

    constructor Create(ADecorateMe :TComponent);

```

end;

А его класс-наследник записывается так.

```
TConcreteDecorator = class (TDecorator)
public
    procedure DoSomeStuff; override;

    constructor Create (ADecorateMe : TComponent);

strict private
    procedure AddedBehavior; // оригинальное поведение
end;
```



Как правило, таких классов-наследников абстрактного Декоратора в реальных проектах создается несколько.

Ключевой процедурой здесь является реализация метода DoSomeStuff.

```
procedure TConcreteDecorator.DoSomeStuff;
begin
    inherited DoSomeStuff;
    AddedBehavior();
end;
```

Здесь выполняется родительская реализация метода DoSomeStuff, расширяющая возможности наследуемого метода. А вызывается на самом деле метод DoSomeStuff объекта FComponent, реализующего базовую функциональность класса TComponent.

```
procedure TDecorator.DoSomeStuff;
begin
    FComponent.DoSomeStuff();
end;

var cc: TComponent;
    dc: TDecorator;
begin

    cc := TConcreteComponent.Create;
    dc := TConcreteDecorator.Create(cc);
    dc.DoSomeStuff;
```

Вызываться же методы будут в таком порядке:

- метод DoSomeStuff, наследуемый объектом dc. В нем имеется прямое обращение к методу FComponent.DoSomeStuff(), который в нашем случае представляет собой метод объекта cc;

- метод `AddedBehavior` объекта `dc`, продолжающий работу метода `DoSomeStuff`.

7.4.5. Шаблон Facade (Фасад)

Фасад продолжает линейку шаблонов, предназначенных для гибкой работы с группами объектов. Он реализует универсальный интерфейс к большой системе, скомпонованной из множества объектов разных классов и взаимосвязей между ними. С точки зрения пользователя Фасада такая система представлена лишь одним интерфейсом (рис. 7.11).

Реализуется этот шаблон очень просто. Имеется класс `TSubsystem`, описывающий поведение подсистемы, представляемой универсальным интерфейсом.

```
TSubsystem = class
public
    procedure SubsystemFunctionality;
end;
```



Обратите внимание, что в этом классе нет никаких ссылок на Фасад — внутренние части подсистемы ничего не знают об охватывающем ее внешнем интерфейсе.

Сам класс фасада:

```
TFacade = class
public
    procedure FacadeMethod;

strict private var
    FTheTSubsystem: TSubsystem;
end;
```

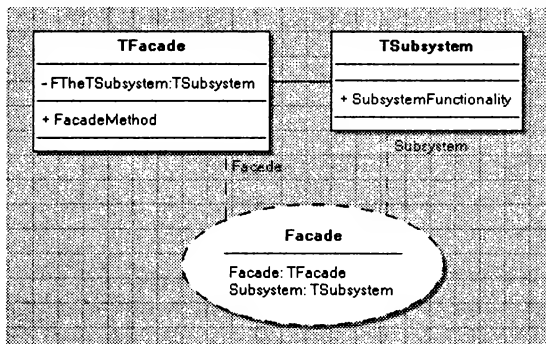


Рис. 7.11. Шаблон Facade в Дизайнере модели

В нем имеется лишь ссылка на подсистему, через которую и происходит управление различными ее функциями. А внешним пользователям предоставляются методы наподобие FacadeMethod, в которых и запрограммированы обращения к тем или иным возможностям подсистемы.

7.4.6. Шаблон Flyweight (Приспособленец)

Приспособленец позволяет существенно повысить эффективность работы приложения в случае, когда задействовано большое число объектов, количество их внутренних состояний ограничено, а основной акцент сделан на функциональные возможности. В таком случае шаблон формирует небольшой набор объектов-состояний, а при создании новых объектов они просто заменяются ссылками на постоянно существующие копии, количество которых равно числу состояний.

Представим, что моделируется футбольная игра, где каждый футболист представлен в виде объекта со сложной логикой поведения, но число состояний которого равно всего двум (выступает за команду А или за команду Б), а вся остальная информация об игре, в том числе и координаты игроков, хранится во внешнем управляющем модуле. Тогда достаточно создавать не 22 объекта (11 объектов команды А и 11 объектов команды Б), а всего два (игрок команды А и игрок команды Б). Внешний программный монитор получит доступ к ним через данный шаблон (рис. 7.12).

Созданием внутренних объектов, отражающих число возможных состояний, занимается класс Фабрика приспособленцев:

```
TFlyweightFactory = class
  strict private var
    FTheTConcreteFlyweights: Hashtable;

  public
    function GetTConcreteFlyweight (AKey :TObject): IFlyweight;
```

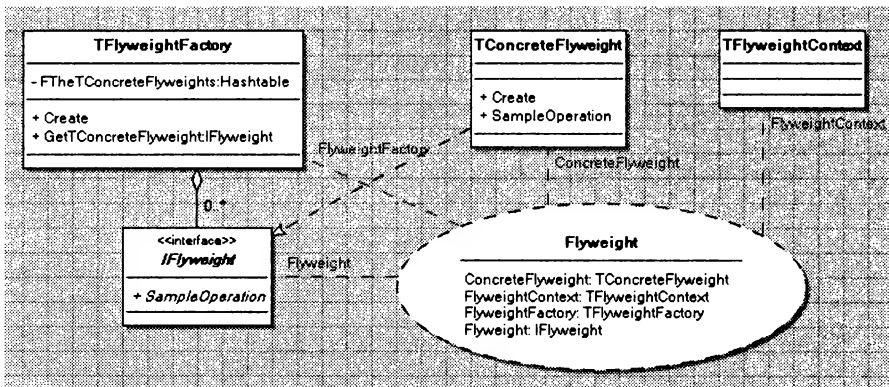


Рис. 7.12. Шаблон Приспособленец в Дизайнере модели

```

constructor Create;
end;

```



Реализация перечня объектов, отображающих возможные состояния, выполнена в данной версии шаблона в Delphi в виде хэш-таблицы — стандартного класса .NET System.Collections.Hashtable.

В программе также возникает дополнительная экономия за счет того, что объект, представляющий конкретное состояние, не создается предварительно, а формируется лишь когда в этом состоянии возникает реальная потребность. Кроме того, не расходуются ресурсы на поддержание полного списка всех состояний, среди которых может быть немало невостребованных.

Шаблон либо возвращает готовый объект, либо создает новый, если интересующее состояние еще не поддерживается. При этом такое действие скрыто внутри Фабрики. Клиентский код обращается лишь к интерфейсу IFlyweight.

```

IFlyweight = interface
  function SampleOperation: String;
end;

```

Этот интерфейс реализуется классом TConcreteFlyweight.

```

TConcreteFlyweight = class(TObject, IFlyweight)
public
  st: String;

  function SampleOperation: String;
  constructor Create(AKey :TObject);
end;

```

На его основе и создаются множественные объекты, доступ к которым формируется на базе их состояний.

Допустим, объект создается на основе ключа, определяющего нужное нам состояние, по которому к нему в дальнейшем будет организован доступ.

```

constructor TConcreteFlyweight.Create(AKey :TObject);
begin
  inherited Create;

  // инициализируем внутреннее состояние:
  st := AKey.ToString;
end;

```

Прикладной метод реализуется, например, так:

```

function TConcreteFlyweight.SampleOperation: String;
begin

```

```

    Result := st;
end;

```

С точки зрения внешнего пользователя всегда происходит обращение только к Фабрике объектов, через метод `GetTConcreteFlyweight` с параметром, отражающим состояние.

```

function TFlyweightFactory.GetTConcreteFlyweight (AKey :
TObject): IFlyweight;
var
    newFlyweight :IFlyweight;
begin
    If FTheTConcreteFlyweights.ContainsKey(AKey) Then
    // если объект уже есть в хэш-таблице:
    begin
        Result := IFlyweight(FTheTConcreteFlyweights[AKey]);
    end
    Else
    begin
    // если объекта нет - создать новый:
        newFlyweight := TConcreteFlyweight.Create(AKey);
        FTheTConcreteFlyweights.Add(AKey, newFlyweight);
        Result := newFlyweight;
    end;
end;

```

В прикладном коде работа с данным шаблоном может выглядеть так.

```

var ff: TFlyweightFactory;
    ifly: IFlyweight;
    i: Integer;
...
ff:= TFlyweightFactory.Create;

for I := 1 to 10 do
    begin
        // создаем формально десять, а фактически - пять объектов
        // в хэше - при создании объектов, в конструкторах которых были
        // указаны одинаковые значения параметра, Приспособленец выполнит
        // подмену реальных объектов ссылками на готовый эталон:
        ifly := ff.GetTConcreteFlyweight(TObject(i div 2));

        label1.Text := ifly.SampleOperation;
    end;

```


7.4.7. Шаблон Proxy (Прокси)

Шаблон Прокси подойдет, когда надо эффективно обработать большое количество компактных объектов. Бывают и обратные случаи, когда объектов в программе немного, но каждый требует больших ресурсов, например если объекты — видеоизображения. В таких ситуациях полезно применять шаблон Прокси. Он предоставляет интерфейс, лишь внешне схожий с оригинальным объектом. Ресурсоемкие операции выполняются только при прямом к ним обращении. В остальных случаях выполняются косметические действия или предоставляется на выполнение код нужных методов. Кроме того, в Прокси-методах обычно встраиваются дополнительные средства контроля, например, за правами пользователя, наличием ресурсов и т.п. (рис. 7.13).

Прокси-интерфейс в данном шаблоне назван `Subject`.

```
Subject = interface
    function SampleMethod(i:Integer): Integer;
end;
```

Этот интерфейс представляет собой оболочку класса `TRealSubject`, который и нуждается в Прокси-расширении.

```
TRealSubject = class(TObject, Subject)
public
    function SampleMethod(i:Integer): Integer;
end;
```

А сам Прокси-класс выглядит так.

```
TProxy = class(TObject, Subject)
    strict private var
```

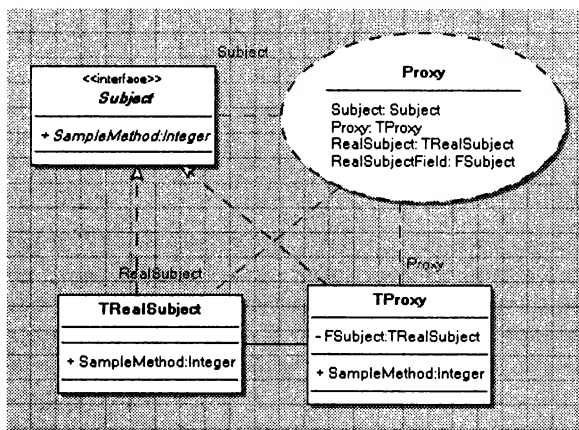


Рис. 7.13. Шаблон Прокси в Дизайнере модели

```
FSubject: TRealSubject;  
  
public  
    function SampleMethod(i: Integer): Integer;  
end;
```

В этом классе имеется ссылка `FSubject` на объект, подлежащий контролю, и функция, «обволакивающая» дополнительным кодом метод оригинального объекта.

```
function TProxy.SampleMethod(i: Integer): Integer;  
begin  
    // дополнительные действия  
    // ...  
  
    result := FSubject.SampleMethod(i);  
end;
```

7.5. Шаблоны поведения

Шаблоны поведения (*Behavioral*) упрощают и проясняют в проекте способы взаимодействия объектов друг с другом. Они формализуют и систематизируют процессы управления внутренней, обычно достаточно сложно организованной работой приложения.

7.5.1. Шаблон Chain of Responsibility (Цепочка обязанностей)

При обработке некоторого запроса внутри программы во многих случаях бывает полезным не завершать его отказом, если конкретный объект-исполнитель не смог выполнить нужные действия (например, был занят или не справился с заданием), а попробовать передать запрос дальше по цепочке — объектам, которые могут предоставить схожие услуги. Если, например, программа, проверяющая орфографию вводимого текста, не может подобрать слово, схожее по написанию с тем, которое ввел человек, то она может ослабить требования к схожести оригинала с предлагаемым вариантом (увеличив число допустимых ошибок в слове) и таким образом расширить область поиска вариантов написания (рис. 7.14).

В шаблоне выделяется класс Обработчик (`THandler`), который принимает сообщение, нуждающееся в неформальной обработке, и распределяет его между объектами, способными обработать запрос. Связь между такими объектами организуется путем наследования возможностей класса `THandler`, а цепочка обслуживания формируется в виде последовательного списка. Сначала создается первый обработчик — экземпляр класса `THandler`, затем с помощью метода `SetSuccessor` указывается следующий объект-обработчик

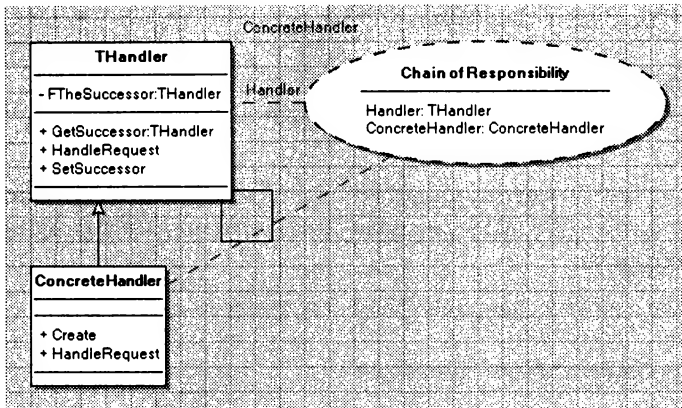


Рис. 7.14. Шаблон Цепочка обязанностей в Дизайнере модели

и так далее. Отметим, что первыми в этом списке надо располагать объекты, способные наиболее точно ответить на сообщение, а далее — менее желательных исполнителей. В результате, когда объект не может обработать сообщение, он просто вызывает соответствующий метод преемника.

type

THandler = class

// ссылка на преемника

strict private var

FTheSuccessor:THandler;

public

// задать преемника

procedure SetSuccessor (ASuccessor :THandler);

// получить преемника

function GetSuccessor: THandler;

// прикладной метод обработки запроса

procedure HandleRequest; **virtual**;

end;

Класс ConcreteHandler наследует возможности класса THandler. Вот как в нем может быть записана реализация метода обработки запроса.

procedure ConcreteHandler.HandleRequest;

var

CanProcessThisRequest :Boolean;

begin

```
// определяем способность обработать запрос:
CanProcessThisRequest := False; // или True, если способны

If CanProcessThisRequest Then
  begin
    // обработка запроса
  end
Else
  begin
    // передаем далее по цепочке
    inherited HandleRequest();
  end;
end;
```

Если теперь подготовить такой код:

```
var c1, c2, c3: ConcreteHandler;
...
c1:= ConcreteHandler.Create(nil);
c2:= ConcreteHandler.Create(c1);
c3:= ConcreteHandler.Create(c2);
...
c3.HandleRequest;
```

то сначала вызовется метод `HandleRequest` экземпляра `c3`, затем, если обработка не выполнится, — метод `HandleRequest` экземпляра `c2`, и в заключение, при необходимости, — метод экземпляра `c1`.

7.5.2. Шаблон Command (Команда)

Шаблон Команда реализует популярную в современном программировании задачу. Он представляет некоторое сообщение в виде объекта, который затем может использоваться заранее неизвестными элементами программы. Такая парадигма широко используется в Delphi. Например, типичные события пользовательского интерфейса (выбор пункта меню, нажатие кнопки, щелчок мышью) в программах обрабатываются заранее не известными способами. Обычно формируется прикладной обработчик события, в который в виде объекта передается информация о событии (рис. 7.15).

В интерфейсе `TCommand` определен метод `Execute`, вызываемый при обращении к объекту, причем заранее не известно — к какому.

```
TCommand = interface
  procedure Execute;
end;
```

На базе интерфейса `TCommand` в клиентском приложении создается класс-наследник `TConcreteCommand` (например, одно из стандартных действий,

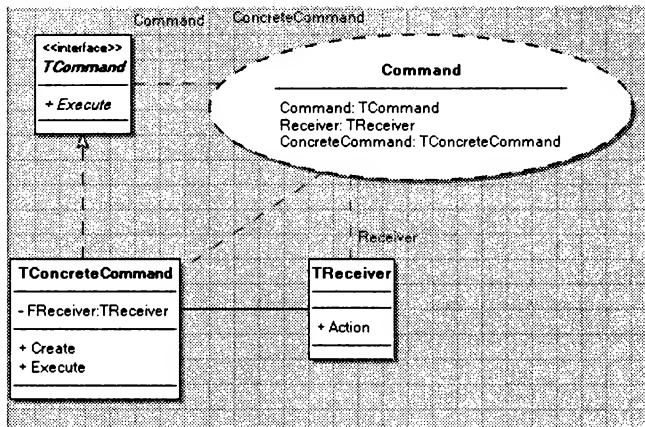


Рис. 7.15. Шаблон Команда в Дизайнере модели

выполняемых при обращении к элементам управления, — реакция на нажатие некоторой кнопки).

```

TConcreteCommand = class(TObject, TCommand)
    strict private var
        FReceiver: TReceiver;

    public
        procedure Execute;
        constructor Create(AReceiver : TReceiver);
end;
  
```

Этот класс используется в объекте Получатель (TReceiver), предоставляющем конкретную реализацию действия Execute. В качестве Получателя может выступать, например, некоторая форма, заинтересованная в обработке нажатия на кнопку.

```

TReceiver = class
    public
        procedure Action;
end;
  
```

В методе Action используется информация объекта — экземпляра TConcreteCommand.

Допустим, в прикладной программе мы создали объект `im`, ответственный за выбор пункта условного меню (не стандартного, а какого-то оригинально запрограммированного в текущем приложении), например Выполнить расчет переменной `A`. Пусть имеется также главный объект (аналог приложения), выполняющий конкретные действия при выборе того или иного пункта меню.

```

var im: TConcreteCommand;
    ap: TReceiver;
...
ap:= TReceiver.Create;
im:= TConcreteCommand.Create(ap);

```

Ключевым здесь является способ реализации метода `Execute`. Если метод `Action` прямо выполняет требуемые от него действия:

```

procedure TReceiver.Action;
begin
// выполнение прикладных действий - расчет переменной A
end;

```

то объект Команда может до или после этого расчета выполнить ряд дополнительных проверок. Например, он может проверить права пользователя и выполнить обработку результата с учетом текущего контекста. Делать это особенно удобно потому, что Команда представляет собой полноценный объект, никак не привязанный к Получателю.

```

procedure TConcreteCommand.Execute;
begin
// действия до ..
    FReceiver.Action();
// действия после ..
end;

```



С помощью данного шаблона удобно создавать метакоманды, не привязанные к конкретным действиям, но поддерживающие типовой набор абстрактных операций: отмену действий, запись макропоследовательностей и так далее.

7.5.3. Шаблон Interpreter (Интерпретатор)

Интерпретатор — специализированный шаблон, выполняющий анализ грамматики некоторого искусственного языка и его интерпретацию. В клиентском коде первоначально формируется некоторое предложение на заранее спроектированном языке, а затем вызывается метод интерпретации (рис. 7.16).

В общем случае класс Абстрактное выражение определяет единственный метод интерпретации выражения, заданного в качестве параметра.

```

TAbstractExpression = class abstract
    public
        procedure Interpret(AContext :TContext); virtual; abstract;
end;

```

Его первый наследник — класс Терминальное выражение — определяет завершающий символ грамматики.

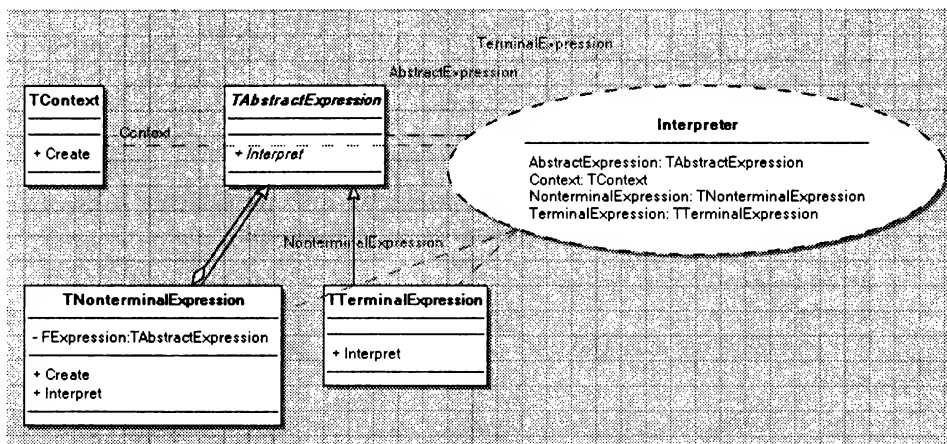


Рис. 7.16. Шаблон Интерпретатор в Дизайнере модели

```

TTerminalExpression = class (TAbstractExpression)
public
    procedure Interpret (AContext : TContext); override;
end;

```

Его второй наследник — класс Нетерминальное выражение — определяет грамматические правила.

```

TNonterminalExpression = class (TAbstractExpression)
    strict private var
        FExpression:TAbstractExpression;

    public
        procedure Interpret (AContext : TContext); override;
        constructor Create (AExpression : TAbstractExpression);
end;

```

Для полноценной работы Интерпретатора необходимо организовать разбор текстовых строк, содержащих выражения, записанные на заданном языке. Конкретные терминальные символы и правила описываются в виде отдельных классов — наследников указанных классов TTerminalExpression и TNonterminalExpression.

7.5.4. Шаблон Iterator (Итератор)

Популярный шаблон Итератор предоставляет программную обложку объектам, содержащим наборы элементов, — спискам, коллекциям. Он предлагает список стандартизованных методов перебора элементов, которые скрывают от конечного пользователя способ своей реализации, за счет чего удается

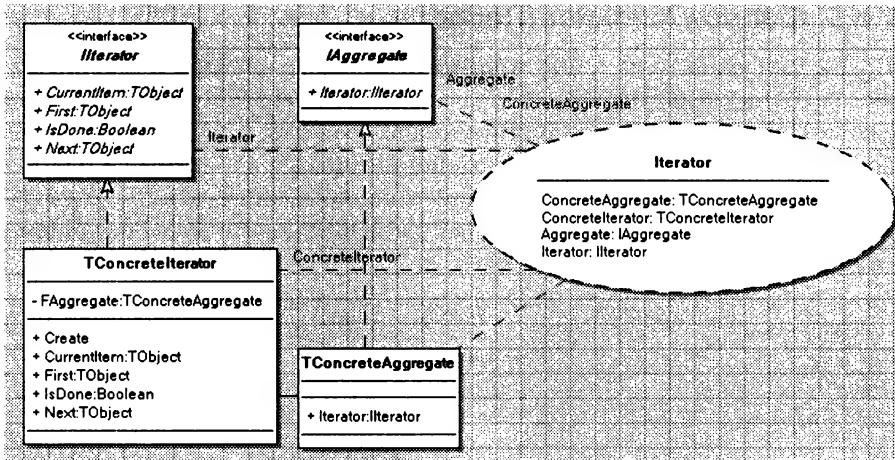


Рис. 7.17. Шаблон Итератор в Дизайнере модели

взаимодействовать с любыми подобными классами через унифицированный интерфейс (рис. 7.17).

Базовые функции Итератора ясны из описания интерфейса:

```

IIterator = interface
    function First: TObject;
    function Next: TObject;
    function IsDone: Boolean;
    function CurrentItem: TObject;
end;

```

Функции First/Next выполняют перемещение к первому и следующему объектам в списке, функция IsDone возвращает значение True, если перебраны все элементы списка, а функция CurrentItem возвращает текущий объект в наборе.

Так как необходимо, чтобы Итератор мог работать с произвольными классами, возможно, встроенными в стандартные системные библиотеки, и предоставлять самую разную функциональность (например, чтобы один Итератор позволял обойти список от начала к концу, а другой — от конца к началу), в программе также требуется некий дополнительный объект, который бы управлял списком доступных итераторов. Поэтому в шаблон включен интерфейс Агрегатор.

```

IAggregate = interface
    function Iterator: IIterator;
end;

```

Агрегатор управляет созданием конкретных классов-итераторов.

7.5.5. Шаблон Mediator (Посредник)

Посредник напоминает некоторые структурные шаблоны. Он позволяет выделить в отдельный объект характерные элементы большой группы объектов. Но он затрагивает не статические свойства, а общее поведение, что позволяет координировать взаимодействие объектов разных классов, например синхронизировать работу элементов пользовательского интерфейса, когда настройки отображения одних элементов связаны с состоянием других (рис. 7.18).

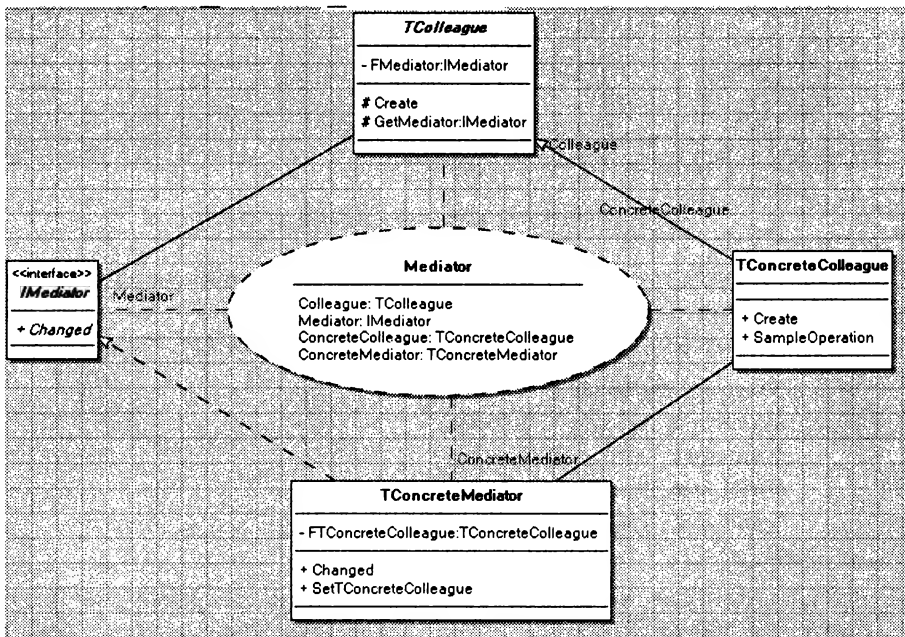


Рис. 7.18. Шаблон Посредник в Дизайнере модели

Интерфейс Посредника содержит основной метод `Changed`, предназначенный для информирования о произошедших изменениях в состоянии некоторого элемента системы.

```
IMediator = interface
    procedure Changed(AColleague :TColleague);
end;
```

Координацией работы элементов группы занимается экземпляр класса-наследника `TConcreteMediator`:

```
TConcreteMediator = class(TObject, IMediator)
    strict private var
        FTConcreteColleague: TConcreteColleague;
```

```

    public
        procedure Changed(AColleague :TColleague);
        procedure SetTConcreteColleague(AColleague :
TConcreteColleague);
    end;

```

Каждый из управляемых классов (Коллег) наследует возможности класса TColleague:

```

TColleague = class abstract
    strict protected
        function GetMediator: IMediator;
        constructor Create(AMediator :IMediator);

    strict private var
        FMediator:IMediator;
end;

```

Коллеги работают только с Посредником, избегая прямого взаимодействия между собой, что и позволяет выделить логику их общения в отдельный объект. Например, конкретная версия класса-коллеги может выглядеть так:

```

TConcreteColleague = class(TColleague)
    public
        procedure SampleOperation;
        constructor Create(AMediator :IMediator);
end;

```

Здесь метод SampleOperation — некоторое прикладное действие, которое должно вызвать реакцию Посредника, и он реализуется примерно так.

```

procedure TConcreteColleague.SampleOperation;
begin
    GetMediator().Changed(self);
end;

```

Метод Changed в классе TConcreteMediator вносит изменения в другие подведомственные классы. Как правило, в конструкторе этого класса формируются элементы управления и запоминаются ссылки на них с последующей синхронной обработкой в методе Changed.

7.5.6. Шаблон Memento (Хранитель)

Хранитель полезен в случаях, когда работа объекта прерывается, например, в исключительных ситуациях или при получении внешнего сигнала, нуждающегося в срочной обработке. Он позволяет не сбрасывать внутреннее состояние объекта, а запомнить его на время отвлечения, чтобы потом продолжить работу с точки прерывания. Подходит Хранитель и для реализации прин-

ципов отката внутреннего состояния системы к некоторым промежуточным контрольным точкам. Еще одна из задач Хранителя — скрытие внутренней информации объекта (рис. 7.19).

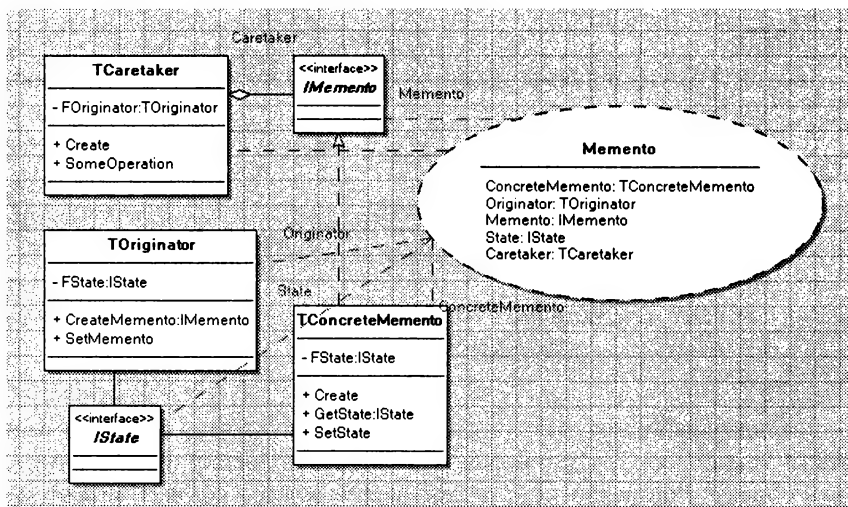


Рис. 7.19. Шаблон Хранитель в Дизайнере объектов

Для реализации подобных механизмов предложена следующая схема (рис. 7.19). Структура конкретного внутреннего состояния описывается в интерфейсе IState. Хозяин TOriginator обращается (изнутри себя) к Хранителю (интерфейс IMemento).

```
Toriginator = class
public
    function CreateMemento: IMemento;
    procedure SetMemento(AMemento :IMemento);

strict private var
    FState:IState;
end;
```

Структура конкретного класса Хранителя такова.

```
TConcreteMemento = class(TObject, IMemento)
public
    procedure SetState(AState :IState);
    function GetState: IState;
    constructor Create(AState :IState);

strict private var
    FState: IState;
end;
```

Получение и сохранение состояний программируется в методах `GetState` и `SetState` через внутреннее поле `FState`.

В клиентской части основное взаимодействие происходит с помощью класса `Посыльный` (`TCaretaker`). Он вызывается, когда нужно сохранить текущее состояние объекта — экземпляра класса `TOriginator`. При этом `Посыльный`, что важно, ничего не знает о внутреннем содержимом состояния `Хранителя`.

```
TCaretaker = class
  public
    constructor Create(AOriginator :TOriginator);

    strict private var
      FOriginator: TOriginator;
end;
```

Работает он с `Хранителем`, например, так:

```
var
  Memento: IMemento;

...

// запрос состояния Хранителя
Memento := FOriginator.CreateMemento();

// нужные действия
...

// возвращение обратно состояния Хранителя
FOriginator.SetMemento(Memento);
```

7.5.7. Шаблон Observer (Наблюдатель)

В современных приложениях, особенно сетевых, популярна возможность синхронного обновления набора объектов при изменении некоторого управляющего объекта. Реализовать такой режим помогает шаблон `Наблюдатель`. При возникновении некоторого события он извещает представителей разных классов о необходимости измениться¹ (рис. 7.20).

Интерфейс `Наблюдателя` `IObserver` содержит единственный метод `Update`, который вызывается внешним управляющим субъектом по мере необходимости.

¹ Иногда такой режим называют «подпиской». Объекты подписываются на автоматическое получение свежих сведений от «издателя».

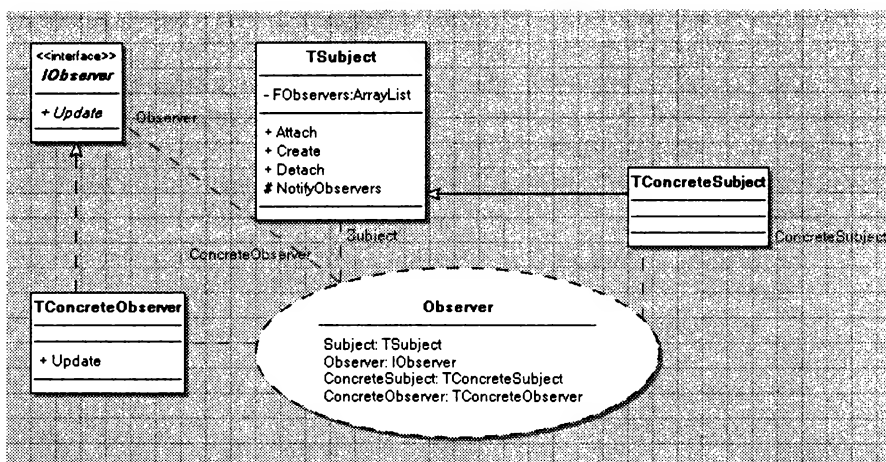


Рис. 7.20. Шаблон Наблюдатель в Дизайнере модели

```

IObserver = interface
    procedure Update(ASubject :TSubject);
end;

```

Класс Субъект содержит список Наблюдателей, которых надо извещать, методы его пополнения и исключения Attach и Detach, а также метод рассылки сообщения всем объектам списка. Отметим, что в качестве параметра такого извещения передается ссылка на сам Субъект.

```

TSubject = class
    strict private var
        FObservers:ArrayList;

    public
        procedure Detach(AObserver :IObserver);
        procedure Attach(AObserver :IObserver);
        constructor Create;

    strict protected
        procedure NotifyObservers;
end;

```

7.5.8. Шаблон State (Состояние)

Шаблон Состояние позволяет выделять элементы поведения объекта, зависящие от его состояния (например, когда среда разработки меняет перечень доступных инструментов в зависимости от текущего объекта, выбранного в Дизайнере). Нередко этот шаблон применяют, чтобы превратить код с большим числом условных операторов и ветвлений в наглядный табличный вид,

когда выбор того или иного действия задается не в коде, а в таблице взаимозависимостей. Фактически, шаблон Состояние выделяет некоторое действие в отдельный класс. Применение его в исходном тексте существенно повышает наглядность программы и позволяет избежать модификации громоздких условных конструкций (рис. 7.21).

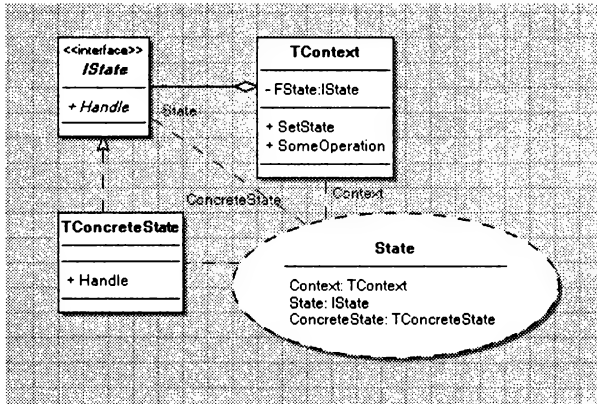


Рис. 7.21. Шаблон Состояния в Дизайнере модели

Интерфейс Состояния прост:

```

IState = interface
  procedure Handle;
end;
  
```

Процедура Handle реализует некоторое действие, которое во внешней программе должно выполняться в зависимости от внутреннего условия.

Управляется Состояние в прикладном коде классом-Контекстом.

```

TContext = class
  public
    procedure SetState(AState :IState);
    procedure SomeOperation;

  strict private var
    FState:IState;
end;
  
```

Экземпляры контекста формируются и настраиваются конкретными состояниями (IState). Он оказывается промежуточным классом, инкапсулирующим внутренние интерфейсы состояний.

Допустим, на форме имеется флажок. В зависимости от его состояния мы хотим вызывать то или иное действие, доступное через интерфейс IState. Для этого создадим два класса-наследника IState — TConcreteState1 и TConcreteState2 — с одинаковой структурой.

```
TConcreteState1 = class(TObject, IState)
    public
        procedure Handle;
    end;
```

В клиентском коде обращение к этим классам может быть таким:

```
var is1, is2: IState;
...
is1 := TConcreteState1.Create;
is2 := TConcreteState2.Create;

if checkbox1.Checked
then is1.Handle
else is2.Handle;
```

Как правило, в качестве названий классов выбирают наглядные имена, чтобы ярче отразить предназначение той или иной ветви. Если эти классы содержат лишь исполнимый код, работа которого никак не связана с их внутренними состояниями, то можно сделать эту запись еще нагляднее:

```
if checkbox1.Checked
then TConcreteState1.Create.Handle
else TConcreteState2.Create.Handle
```

7.5.9. Шаблон Strategy (Стратегия)

Этот весьма полезный в прикладной разработке шаблон позволяет изменять реализацию некоторого алгоритма без модификации клиентского кода. Это осуществляется предоставлением доступа к функциям через универсальный интерфейс, не зависящий от реализации. Обычно программисты решают эту проблему выделением исполнимого кода в библиотеку, отчуждаемую от программы (например, в *DLL*-файл, который можно заменять), однако такой подход не годится, если изменилась сама форма обращения к функции (например, появились новые параметры или возникло новое название). Шаблон Стратегия также поможет, если в программе имеется много условных ветвлений (частный случай — выбор реализации алгоритма в зависимости от значения параметра) — рис. 7.22.

В интерфейсе *IStrategy* описывается конкретный перечень функций, подлежащий отчуждению от их реализации. Пусть он будет таким.

```
IStrategy = interface
    function Sample: integer;
end;
```

А реализуется этот интерфейс классом *TConcreteStrategy*:

```
TConcreteStrategy = class(TObject, IStrategy)
```

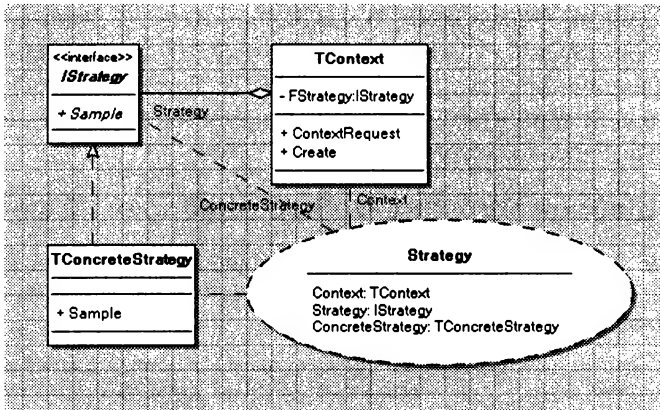


Рис. 7.22. Шаблон Стратегия в Дизайнере модели

```

public
    function Sample: integer ;
end;

function TConcreteStrategy.Sample: integer ;
begin
    result := 1;
end;

```

Работа с этим интерфейсом поддерживается Контекстом — классом TContext:

```

TContext = class
    strict private var
        FStrategy: IStrategy;

    public
        function ContextRequest: Integer ;
        constructor Create(AStrategy : IStrategy);
end;

```

При конструировании контекст получает в качестве параметра экземпляра одного из наследников интерфейса IStrategy:

```

var cx: TContext;
...
cx:= TContext.Create( TConcreteStrategy.Create );

```

Теперь к соответствующему методу можно обратиться так:

```

label1.Text := cx.ContextRequest.ToString;

```


При изменении алгоритма функции `Sample` достаточно лишь модифицировать ее реализацию для класса `TConcreteStrategy`, не внося изменения в существующую структуру классов и их взаимосвязи. И расширение перечня реализаций алгоритма `Sample` также происходит прозрачно, добавлением нового класса-реализатора `IStrategy`.

7.5.10. Шаблон Template Method (Шаблонный метод)

Этот шаблон расширяет возможности шаблона Стратегия и отличается от него тем, что позволяет заменять не весь алгоритм целиком, а лишь отдельные его шаги (рис. 7.23).

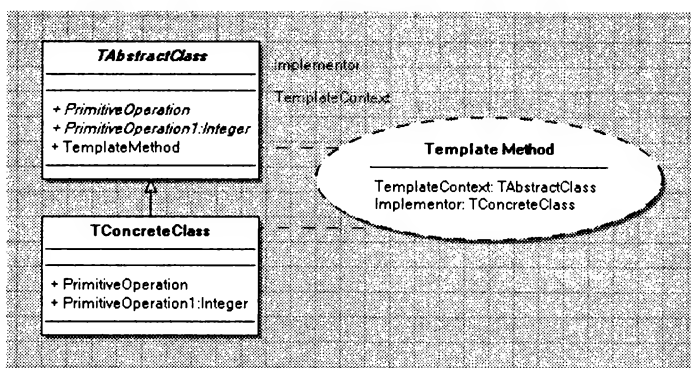


Рис. 7.23. Шаблонный метод в Дизайнере модели

Список элементов алгоритма, подлежащих замене, задается в интерфейсе `TAbstractClass`:

```
type
  TAbstractClass = class abstract
  public
    procedure TemplateMethod;
    function PrimitiveOperation1: Integer; virtual; abstract;
    procedure PrimitiveOperation; virtual; abstract;
  end;
```

Процедура `TemplateMethod` — основной, неизменяемый алгоритм. Она составлена из последовательности вызова переопределяемых методов:

```
procedure TAbstractClass.TemplateMethod;
begin
  PrimitiveOperation();
  ...
  PrimitiveOperation1();
end;
```

Наследуется этот класс, например, одной из возможных реализаций алгоритма.

```
TConcreteClass = class(TAbstractClass)
  public
    function PrimitiveOperation1: Integer; override;
    procedure PrimitiveOperation; override;
  end;
```

Переопределять каждый раз все методы не обязательно — можно ведь наследовать и их реализацию из базового класса TAbstractClass, а модификации подвергать лишь отдельные шаги.

7.5.11. Шаблон Visitor (Посетитель)

Последний из рассматриваемых здесь шаблонов, Посетитель, позволяет выделить некоторый набор операций, характерный для множества классов с различными структурами. Этот набор по различным причинам нежелательно вносить в описание классов (например, он может постоянно расширяться, а число различных классов, подлежащих модификации, велико), и в то же время вызываться должны методы, четко привязанные к конкретным контекстам исполнения (рис. 7.24).

Изложенная схема реализуется с помощью шаблона Посетитель следующим образом. Существует интерфейс IElement:

```
IElement = interface
  procedure Accept(AVisitor :IVisitor);
end;
```

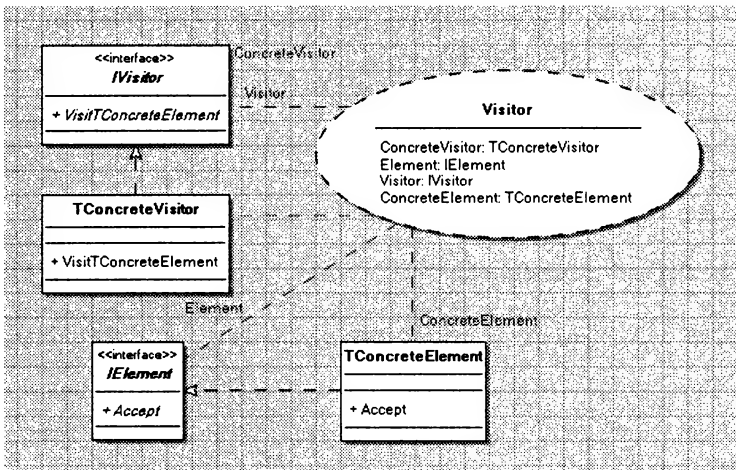


Рис. 7.24. Шаблон Посетитель в Дизайнере модели

Он поддерживается множеством существующих классов. Основной его метод `Accept` передает внутрь класса ссылку на объект-Посетитель, ответственный за реализацию контекстно-зависимой операции, вынесенной в то же время в независимый интерфейс `IVisitor`.

```
IVisitor = interface
  procedure VisitTConcreteElement (AElement :
TConcreteElement);
end;
```

Здесь процедура `VisitTConcreteElement` в свою очередь получает ссылку на конкретный класс как параметр, необходимый для доступа к его состоянию.

Реализация метода `Accept` представляет собой вызов нужного метода конкретного класса, поддерживающего интерфейс `IElement`.

```
TConcreteElement = class(TObject, IElement)
  public
    procedure Accept(AVisitor :IVisitor);
end;

procedure TConcreteElement.Accept(AVisitor :IVisitor);
begin
  AVisitor.VisitTConcreteElement(self);
end;
```

Пусть имеются два класса: `TConcreteElement` и `TConcreteElement2`, каким-то образом различающиеся по внутреннему описанию, но поддерживающие интерфейс `IElement`. Для каждого из них в интерфейсе `TVisitor` описывается свой метод.

```
IVisitor = interface
  procedure VisitTConcreteElement (AElement :
TConcreteElement);
  procedure VisitTConcreteElement2 (AElement :
TConcreteElement2);
end;

procedure TConcreteElement.Accept(AVisitor :IVisitor);
begin
  AVisitor.VisitTConcreteElement(self);
end;

procedure TConcreteElement2.Accept(AVisitor :IVisitor);
begin
  AVisitor.VisitTConcreteElement2(self);
end;
```

В клиентском коде работа с такой структурой обычно происходит путем перебора всех или некоторого подмножества экземпляров классов, поддерживающих интерфейс `IElement`, для каждого из них с вызовом метода `Accept`:

```
var cel: TConcreteElement;  
    ce2: TConcreteElement2;  
    iv: IVisitor;  
...  
  
iv := TConcreteVisitor.Create;  
cel:= TConcreteElement.Create;  
ce2:= TConcreteElement2.Create;  
  
// здесь реализуется перебор объектов cel, ce2, ..., ceN:  
  
cel.Accept(iv);  
ce2.Accept(iv);
```

Какие конкретно методы вызываются, в клиентском коде понять нельзя благодаря инкапсуляции. В дальнейшем перечень операций, допустимых над объектами (в дополнение к единственному пока методу `Accept`), можно расширять, не внося в дополнительный код уже существующие классы.

7.6. Обслуживание шаблонов

Мы рассмотрели все шаблоны, относящиеся к каноническому набору GoF. При желании перечень доступных шаблонов Delphi можно расширить самостоятельно. Для этого надо подготовить в проекте UML-модель, выделить на ней нужную группу элементов (например, с нажатой клавишей `CTRL`) и затем дать команду контекстного меню `Save As Pattern` (Сохранить как шаблон). В открывшемся диалоговом окне, в поле `Name` (Имя) указывается название нового шаблона, в поле `File` (Файл) — название XML-файла, в котором сохранится его описание, в поле `Description` (Описание) — описание шаблона. На следующем шаге создания (кнопка `Next` (Далее)) задаются параметры шаблона — их список будет автоматически построен с учетом элементов модели, включенных в данный шаблон. Наконец, на следующем шаге выбирается группа шаблонов, к которой будет отнесен новый шаблон. Впоследствии он будет доступен так же, как и остальные шаблоны, посредством панели инструментов `Node by Pattern/Link by Pattern`.

Система Delphi допускает корректировку структуры шаблонов. Для этого надо воспользоваться Диспетчером шаблонов, который вызывается командой главного меню `Tools > Pattern Organizer` (Сервис > Диспетчер шаблонов) — рис. 7.25.

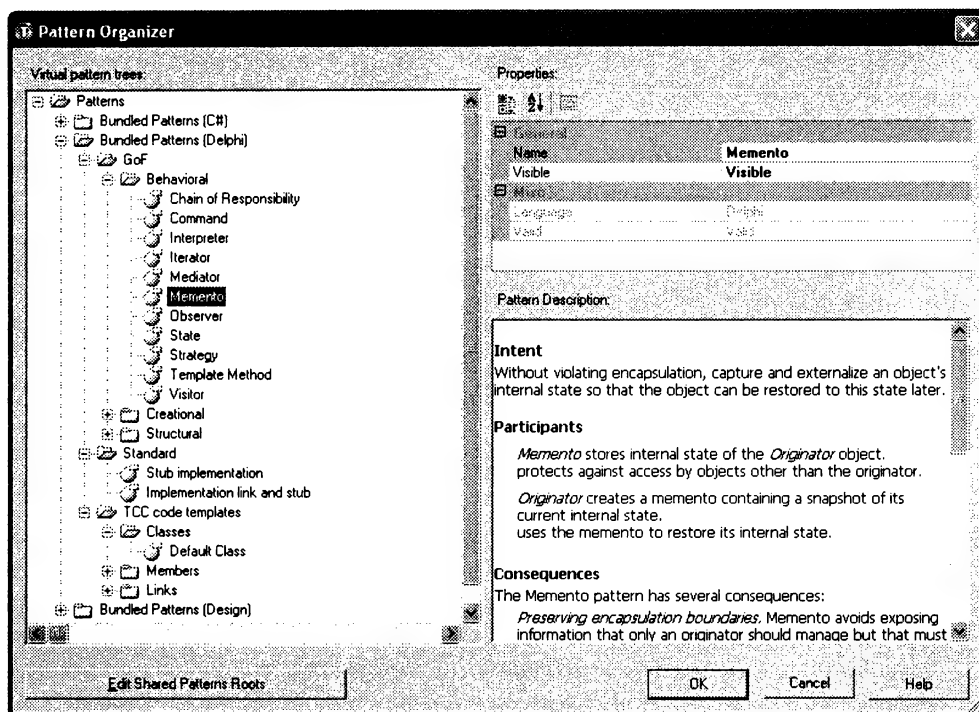


Рис. 7.25. Диспетчер шаблонов

Шаблоны распределены по языкам (C#, Delphi и UML), а собственные шаблоны обычно добавляются в группу Custom Patterns. Здесь можно и удалить любой шаблон, просто выбрав его в дереве шаблонов и выполнив команду Delete (Удалить) контекстного меню.

Глава 8. Технология моделирования с помощью языка UML

8.1. Унифицированный язык моделирования

Унифицированный язык моделирования Unified Modeling Language (UML) предназначен для использования в объектно-ориентированных системах разработки, таких, как Delphi. Он позволяет уменьшить разрыв между этапом проектирования системы, построения ее архитектуры и внутренних взаимосвязей и этапом кодирования. Технология UML дает возможность построить абстрактную модель системы, не привязанную к конкретному языку программирования. Она опирается на набор понятий и концепций, которые доступны специалисту, не знакомому с программированием. Кроме того, визуальные средства UML позволяют уже на первых этапах подготовки требований к системе привлекать представителей заказчика, вообще не знакомых с информационными технологиями. Это достигается за счет наглядной и выразительной графической нотации.

Технология UML базируется на трех фундаментальных понятиях:

- сущность — объект проектируемой системы, который возможно достаточно целостно выделить в абстрактном виде;
- отношение — способ и форма связи между сущностями;
- диаграмма — визуальное представление комплекса объектов с отношениями между ними, выраженное с помощью набора предопределенных графических элементов.

В системе Delphi 2006 поддерживаются две версии UML: версия UML 1.5, которая насчитывается восемь классов диаграмм (диаграммы классов, диаграммы вариантов использования, диаграммы последовательности, диаграммы состояний, диаграммы кооперации, диаграммы деятельности, диаграммы компонентов и диаграммы развертывания), и версия UML 2.0, в которой всего тринадцать типов диаграмм, из которых Delphi поддерживает девять¹.

¹

Из отличий отметим диаграммы компонентов, диаграммы внутренней структуры, диаграммы машин состояний и диаграммы взаимодействия.

8.2. Составные части диаграмм

Сущности и отношения представляются на диаграммах в графическом виде. При этом, конечно, востребованы и различные текстовые надписи, и сопроводительные комментарии, поэтому диаграммы составляются из элементов четырех типов.

1. Двумерные графические объекты, состоящие, обычно, из простейших геометрических фигур, которые применяются для изображения сущностей. Эти объекты могут менять свои размеры: растягиваться или сжиматься. Например, прямоугольник представляет собой некоторый класс программы. В зависимости от числа таких классов, размещенных в пространстве моделирования, некоторые из них могут быть сделаны большими, другие — меньшими. Одни из объектов могут размещаться внутри других.
2. Пути — последовательности отрезков, соединяющие графические объекты. С помощью путей, например, демонстрируется схема наследования между классами.
3. Символы или значки, размеры и формы которых не меняются, вкладывание друг в друга не допускается. Чаще всего символами являются стрелочки на концах линий связи.
4. Текстовые надписи, связанные по смыслу с некоторой сущностью. Например, элементы класса (поля, методы) могут представляться названиями внутри прямоугольника.

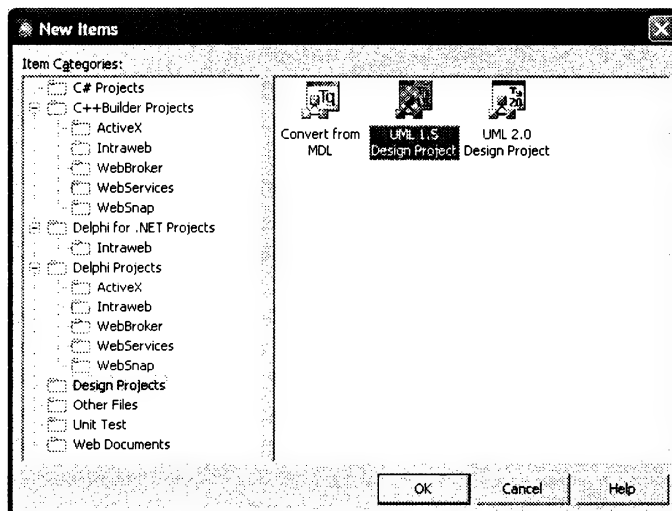


Рис. 8.1. Создание пустого UML-проекта

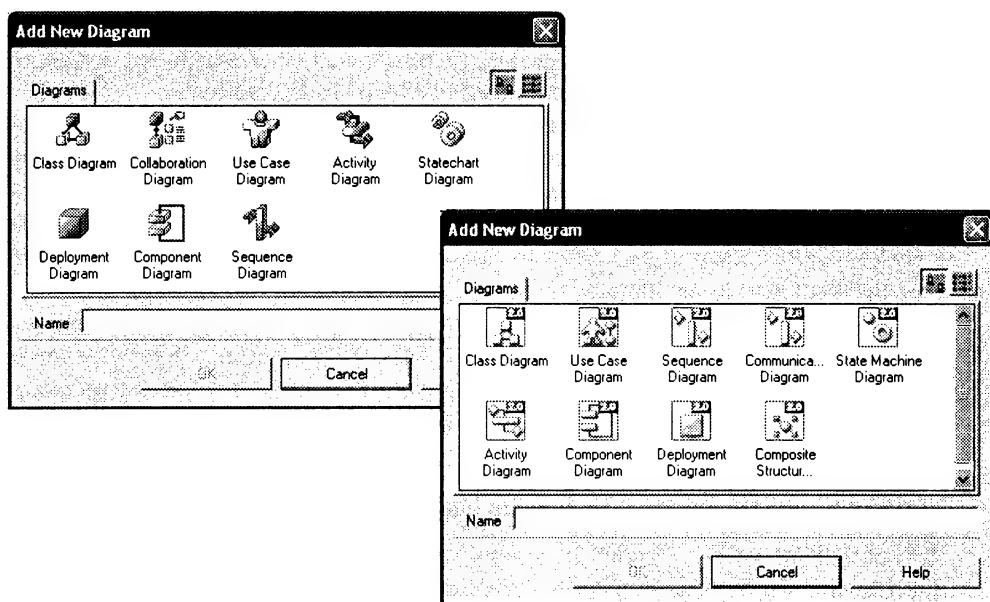


Рис. 8.2. Восемь диаграмм UML 1.5 против девяти диаграмм UML 2.0

8.3. Создание UML-проекта

Чтобы создать пустой UML-проект, надо дать команду главного меню File ➤ New ➤ Other (Файл ➤ Создать ➤ Прочее) и в разделе Design Projects (Моделирование) выбрать значок UML 1.5 Design Project, если создается диаграмма в стандарте UML 1.5, или UML 2.0 Design Project, если создается диаграмма в стандарте UML 2.0 (рис. 8.1).

После выбора стандарта в Просмотрщике объектов появится пустая заготовка модели. Если в нем дважды щелкнуть на строке с названием проекта, откроется пустое модельное пространство. На нем размещаются ссылки на диаграммы конкретных типов. Добавляется любая из доступных диаграмм командой Add ➤ Other Diagram (Добавить ➤ Иную диаграмму) и последующим выбором нужного значка в списке доступных диаграмм. Перечни доступных диаграмм для стандартов UML 1.5 и UML 2.0 несколько различаются (рис. 8.2).

8.4. Технология моделирования UML 1.5

8.4.1. Диаграмма классов (Class Diagram)

Наиболее важными и распространенными среди программистов по праву считаются *диаграммы классов (class diagram)*. С их помощью описываются классы, используемые в программе, представляются схемы наследования,

определяются поля и методы. Принципы описания классов средствами языка UML понятны любому программисту, знакомому с объектно-ориентированным программированием и схемами наследования классов.

Главная особенность диаграмм классов заключается в том, что они описывают статическую, неизменную структуру приложения. Временные характеристики, последовательности выполнения тех или иных логических операций в них не рассматриваются. Поэтому диаграммы классов задействованы в технологиях Delphi и ESO в качестве основного инструмента моделирования, привычного подавляющему числу разработчиков.

В языке UML существует понятие класса, практически эквивалентное понятию класса в объектных языках программирования. Оно определяет структуру, шаблон абстрактного понятия, на основе которого в процессе работы программы может создаваться множество одинаковых по своей структуре объектов.

Добавление класса на диаграмму Delphi

Диаграмму классов создают командой контекстного меню с пространства диаграммы Add ➤ Other Diagram ➤ Class Diagram (Добавить ➤ Другую диаграмму ➤ Диаграмму класса (рис. 8.3).

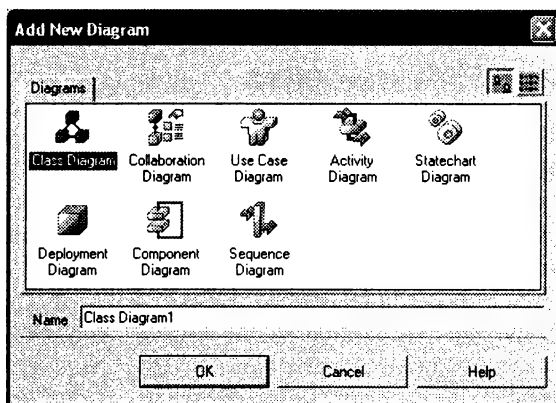


Рис. 8.3. Создаем диаграмму классов

Указанная команда создает пустое пространство моделирования. Добавить к нему класс можно, выбрав на панели инструментов пункт Class (Класс) и щелкнув мышью в подходящей точке Дизайнера. В появившемся элементе-классе надо ввести новое название класса или оставить то, которое предлагается по умолчанию — Class1.

На диаграмме классов класс представляется прямоугольником (рис. 8.4). Внутри него горизонтальными линиями могут быть выделены разделы. В первом разделе всегда указывается имя класса. Допускаются представления класса, состоящие только из одного его имени. В таких случаях специальные разделы внутри соответствующего прямоугольника не формируются.

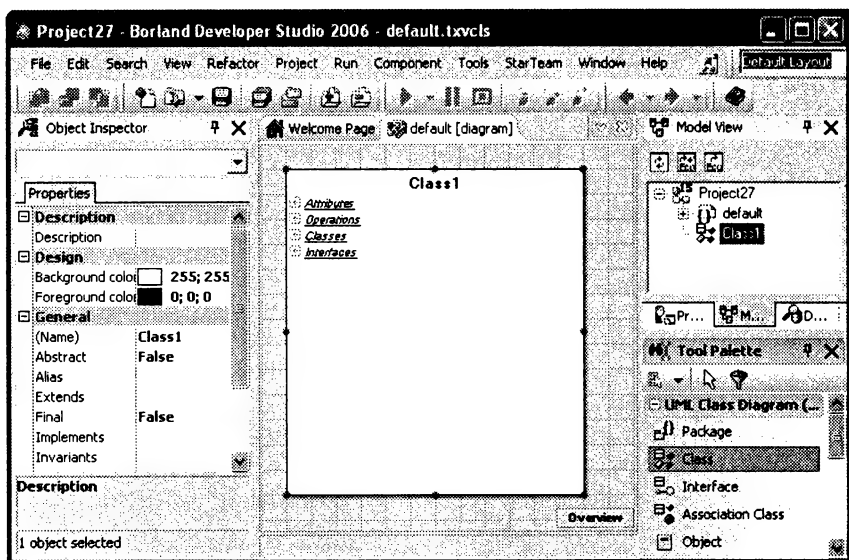


Рис. 8.4. Представление класса на диаграмме классов

Во втором разделе задается перечень свойств класса (полей, атрибутов), в третьем — методы (операции), в четвертом, необязательном, — исключения. Но если указывается второй или третий раздел, то их должны дополнять соответственно третий и второй, даже если их содержимое пусто или пока не определено.

Имя класса записывается полужирным шрифтом, располагается по центру и начинается с прописной буквы. Конечно, на одной диаграмме не должно быть классов с одинаковыми названиями.

Если класс абстрактный (не допускается создавать его экземпляры), то имя его указывается наклонным шрифтом. Абстрактный класс определяется значением `True` свойства `Abstract` в Инспекторе объектов.

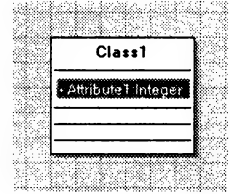
Классы, предназначенные для определенных специфических нужд, можно настраивать с использованием свойства `Stereotype` (стереотип восприятия) — важнейшего свойства многих элементов разных диаграмм UML, уточняющих конкретное назначение элемента. Например, значение `enumeration` стереотипа задает элемент известного в Delphi типа *перечисление* (список абстрактных значений ограниченного диапазона).

Свойства/атрибуты класса

Свойство класса (в UML оно называется *атрибутом*) добавляется выделенному на диаграмме классу командой контекстного меню `Add > Attribute` (Добавить > Атрибут). Если в свойстве `static` атрибута установлено значение `True`, то соответствующий атрибут считается статическим, принадлежащим непосредственно классу, а не его экземпляру.

Значение атрибута может быть неизменным на протяжении всего времени существования экземпляра класса¹. Для этого надо свойству `ReadOnly` атрибута задать значение `True`.

Атрибуты, располагающиеся во втором разделе класса, записываются по следующему универсальному шаблону (порядок в зависимости от среды реализации может немного меняться):



описатель-видимости имя-свойства :
тип-свойства кратность = значение { константа }.

Обязательным в этом шаблоне является лишь имя свойства.

Описатель видимости (свойство `Visibility` атрибута в Инспекторе объектов) задает видимость поля. Он принимает следующие значения:

- «+» — свойство видимо всем (`public`);
- «#» — свойство видимо лишь в данном классе и классах-наследниках (`private`);
- «-» — свойство видимо лишь в пределах данного класса (`private`);
- «~» — свойство видимо в пределах текущего пакета (`package`).



Если описатель не указан, то видимость поля не определена.

Кратность (свойство `Multiplicity` атрибута в Инспекторе объектов) определяет допустимое число свойств данного типа в текущем классе. Кратность записывают в квадратных скобках в форме, напоминающей запись множеств Паскаля, например:

```
[ 0..1 ]
[ 2..5, 7, 9..20 ]
```

Допускается также использование символа «*» (любое целое неотрицательное число).

Кратность часто применяется для задания отношений между наборами, которые характерны при разработке структур баз данных (один-ко-многим, многие-ко-многим и другие). Она также может задавать количество элементов объекта (как массив).



По умолчанию кратность равна 1.

Тип свойства обычно определяется конкретной реализацией среды моделирования. Практически всегда среди типов свойств можно найти `integer` (целое число), `string` (строку) и `bool` (логическое значение). Вслед за типом после сим-

¹ В языке UML 1.x для этих целей нередко применяется свойство `frozen` (замороженный), удаленное из UML 2.0.

вола « $\Rightarrow$ » может следовать начальное значение, диапазон которого определяется типом свойства. Это значение вводится в свойстве InitialValue.



Название свойства может быть подчеркнуто. Это означает, что такое свойство способно принимать не одно значение, а некоторое множество значений (тип данных set в Delphi).

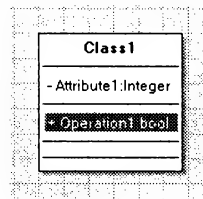
Если начальное значение взято в фигурные скобки, это означает, что значение соответствующего свойства недопустимо изменять в процессе выполнения программы (фактически, оно считается константой):

```
Программист : salary = { $150 }
```

Методы

Метод (в языке UML он называется *операцией*) добавляется выделенному на диаграмме классу командой контекстного меню Add ➤ Operation (Добавить ➤ Операция).

Универсальная структура описания метода в диаграмме UML такова:



```
описатель-видимости имя-метода (список-параметров) :
тип-возвращаемого-значения { свойства-метода }
```

Описатель видимости аналогичен соответствующему элементу свойства класса. Список параметров составлен из описаний конкретных параметров, разделенных запятыми. Каждый параметр, в свою очередь, записывается так:

```
вид-параметра имя-параметра : тип-параметра = значение-по-
умолчанию
```

В качестве вида параметра может использоваться одно из следующих ключевых слов:

- in (значение параметра передается в метод и не может быть модифицировано);
- out (значение параметра формируется внутри метода и доступно после завершения работы метода);
- inout (параметр можно использовать как для передачи значения внутрь метода, так и для получения значения из него).



Если задано значение параметра по умолчанию, то оно автоматически присваивается параметру метода перед вызовом последнего.

Список параметров метода формируется обращением к его свойству Parameters, в результате чего вызывается диалоговое окно, в котором редактируется перечень сформулированных параметров. Добавляется новый параметр кнопкой Add (Добавить); при этом его настройка выполняется в дополнительном диалоговом окне (рис. 8.5).

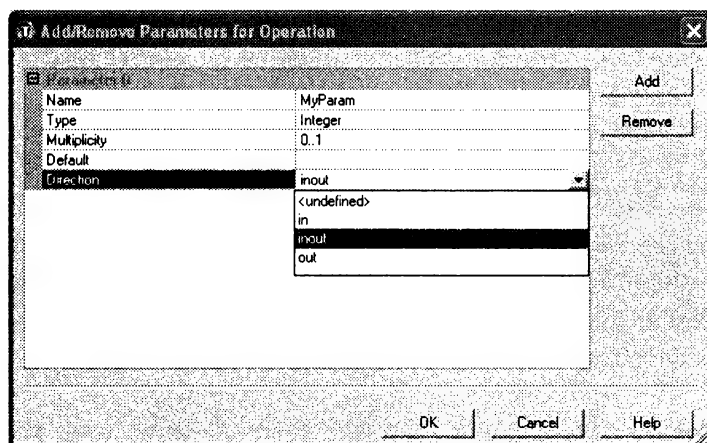


Рис. 8.5. Настройка способа передачи параметра в метод

Метод можно явно сделать абстрактным, записав в его свойстве **Abstract** значение **True**, при этом название метода будет выделено курсивом.

В общем случае метод может выполняться разными способами: как полностью захватывая доступные ресурсы системы, так и разделяя их с другими процессами. Эту характеристику можно задать (в описании метода, в свойстве **Description**) в формате:

```
{concurrency=способ-исполнения}
```

Здесь способ исполнения принимает одно из следующих значений:

- **sequential** — метод должен выполняться одиночно;
- **concurrent** — метод может выполняться одновременно с другими методами;
- **guarded** — при вызове данного, возможно потенциально опасного или неотлаженного метода надо обратить особое внимание на обработку исключительных ситуаций.

Отношения

Различные объекты (сущности) на диаграмме могут связываться друг с другом в различных отношениях. В UML существует четыре типа отношений: ассоциации (**association**), зависимости (**dependency**), обобщения (**generalization**) и реализации (**realization**).



Отношения реализации существуют лишь для диаграмм компонентов UML 2.0 и рассматриваются в соответствующем разделе.

Отношение ассоциации. Отношение ассоциации задает некоторую взаимосвязь между классами. Оно обычно представляется сплошной синей линией, соединяющей два класса (рис. 8.6).



Рис. 8.6. Отношение ассоциации

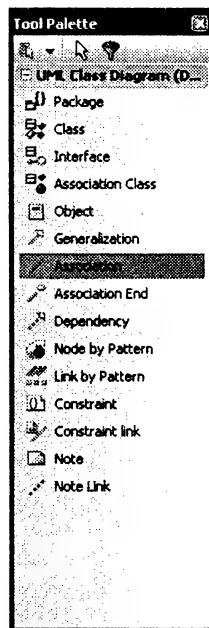
Для установления ассоциации между классами служит инструмент Association палитры инструментов. Сначала выполняется щелчок на одном классе, затем протягивается линия к другому классу и на нем выполняется второй щелчок.

Отношение ассоциации также может дополняться:

- названием отношения (свойство Label в Инспекторе объектов); записывается с прописной буквы и располагается рядом с линией отношения;
- порядком следования классов (свойство Directed), первый из которых считается клиентом, а второй — поставщиком. Направленное отношение отображается линией со стрелкой, ведущей от первого класса ко второму.

Нередко узлы ассоциации помечаются указанием роли, которую играет в ассоциации класс, расположенный в данном узле. Для этой цели служат свойства Client Role (Роль клиента) и Supplier Role (Роль поставщика).

На концах ассоциаций также указывается кратность класса, задающая число экземпляров данного класса, если количество экземпляров других классов данной ассоциации уже известно. Кратность указывается в свойствах Client Cardinal (Кратность клиента) Supplier Cardinal (Кратность поставщика).



Отношение агрегации. Отношение агрегации, как частный случай отношения ассоциации, задает связь часть-целое, когда один из элементов диаграммы представляет собой составную часть другого элемента. Классы, один из которых обозначает целое, а несколько других — его части, связываются сплошными линиями. Окончание линии связи, ведущее к классу-целому, отмечается ромбом. Для формирования подобной связи надо выделить связь-ассоциацию и задать в ее свойстве Type (Тип) значение aggregation (агрегация). При этом «целым» считается класс, находящийся с клиентской стороны соединения (рис. 8.7).

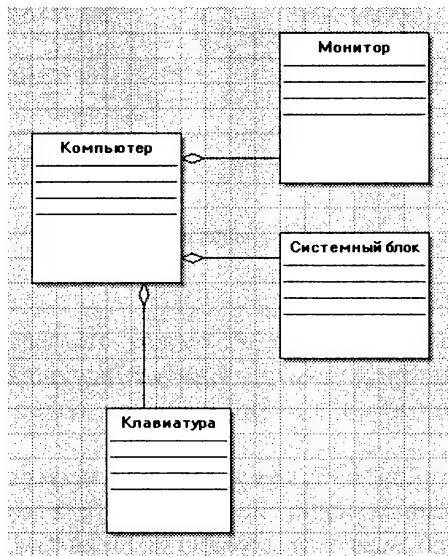


Рис. 8.7. Пример отношений агрегации

Так, настольный компьютер (фактически, некое условное понятие) физически может состоять из системного блока, монитора и клавиатуры, причем эти составляющие могут легко заменяться.

Отношение композиции. Отношение композиции, в свою очередь, является частным, более строгим случаем отношения агрегации. Оно задает связь между целым и такими его частями, которые не могут существовать отдельно от целого. В этом случае окончание линии отношения обозначается закрашенным ромбом.

Формируется отношение композиции вводом в свойстве Type (Тип) связи-ассоциации значения *composition* (композиция) — рис. 8.8.

Компьютер состоит из материнской платы, процессора и оперативной памяти, которые являются его неотъемлемыми внутренними частями. После изъятия любой из них будет полностью неработоспособен.

Классы ассоциаций. Ассоциации — столь мощный механизм, что в ряде случаев бывает желательно уточнить настройки связи, детализировать ее поведение с помощью свойств и методов, например для добавления характеристик силы связи, ее дополнительных функций. Для этого применяются так называемые классы-ассоциации (*Association Class*).

Класс, представляющий ассоциацию, соединяется с двумя исходными классами, связанными ассоциативной связью, через элемент ромбической формы (связь основных классов с ромбом формируется элементом *Association End*, который проводится от ромба к классу). В свою очередь, класс, реализующий ассоциацию, связывается с ромбом пунктирной линией.

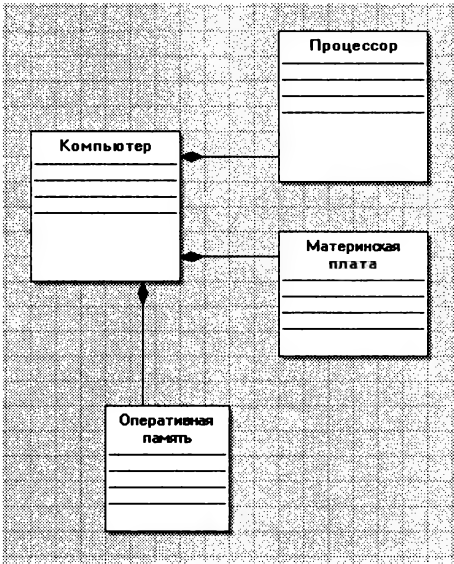


Рис. 8.8. Композиция — частный случай агрегации

Например, на рис. 8.9. показано отношение между исполнителем и заказчиком через связь-ассоциацию, которая представляет класс Контракт, содержащий множество подробных сведений об этом отношении.

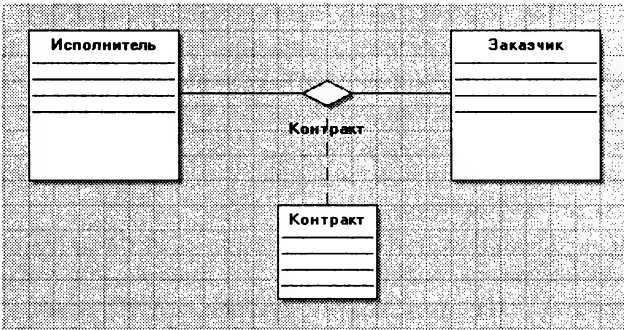


Рис. 8.9. Связь через класс ассоциации

Отношение обобщения. Этот распространенный вид отношений применяется, например, для визуального построения иерархии классов. Отношение представляется пунктирной линией со стрелкой на одном из ее концов (элемент Generalization) — рис. 8.10.

Направление связи всегда задается от обобщаемого класса к обобщающему, например от класса-наследника к классу-родителю. У более общего класса может быть несколько обобщаемых классов.

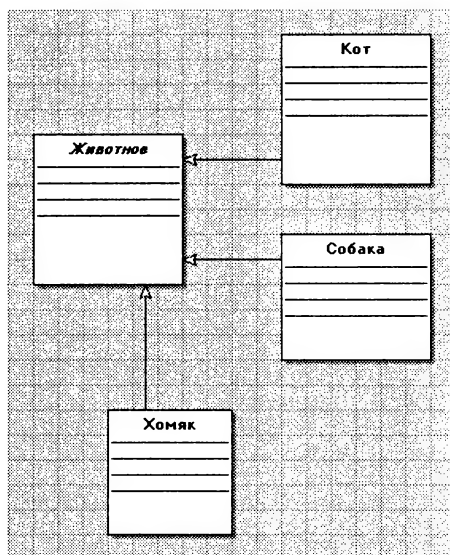


Рис. 8.10. Пример связей обобщения



Множество линий со стрелками, ведущими к наиболее общему классу, допускается заменять одной линией со стрелкой и общей точкой разветвления.

Рядом со стрелкой может быть указана надпись (свойство Label связи), в которой уточняется ограничение отношения, распространяемое на все обобщаемые классы. Оно может принимать одно из следующих значений:

- {complete} — в отношении указаны все классы-наследники, и других у класса-родителя быть не может;
- {incomplete} — в отношении указаны не все классы-наследники (например, потому что их очень много);
- {disjoint} — не допускаются реализации, охватывающие функциональность нескольких классов-потомков, помеченных как disjoint;
- {overlapping} — явно допускаются реализации, охватывающие функциональность нескольких классов-потомков, помеченных как overlapping.

Часто родительский класс делают абстрактным, чтобы продемонстрировать абстрактные и реально используемые элементы иерархии.

Отношения зависимости. Этот тип отношений (элемент Dependency палитры инструментов) определяет общий случай смысловой зависимости между двумя элементами. Он задается пунктирной линией со стрелкой особой формы (рис. 8.11).

Отношение зависимости направлено от класса-клиента к классу-источнику. Его часто применяют, чтобы указать, что некоторые методы класса используют в качестве параметров экземпляры других классов. Например,

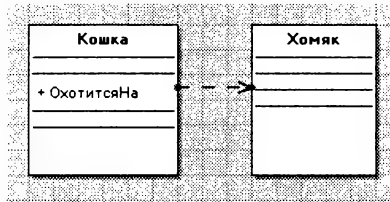


Рис. 8.11. Пример отношения зависимости

метод `ОхотитсяНа(X: Хомяк)` класса `Кошка` использует параметр с типом `Хомяк` (другой класс проекта).

Интерфейсы

Интерфейсы (пункт `Interface` палитры инструментов) соответствуют понятию *Interface* языка Delphi. Интерфейсы служат лишь описаниями и не требуют явного наличия своей реализации. Эта популярная концепция программирования выполнена в UML в виде графического элемента, который по внешнему виду напоминает элемент класса, только над его названием указывается слово `interface` (в угловых скобках), а из разделов доступны лишь раздел свойств и раздел методов (рис. 8.12).

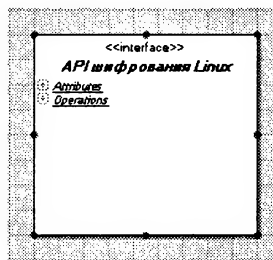


Рис. 8.12. Графический элемент *Interface*

Объекты

В ряде диаграмм UML используются не только классы, но и объекты — реально существующие в программе экземпляры, конкретные представители классов. Изображение объекта отличается от изображения класса:

- название объекта всегда подчеркивается;
- название объекта составлено из названия экземпляра объекта, двоеточия и названия класса объекта. Если название экземпляра отсутствует, то указывается лишь название класса с префиксом-двоеточием. Если название класса отсутствует, то указывается лишь название объекта, а двоеточие опускается (рис. 8.13).

Объекты добавляются на диаграмму инструментом `Object` палитры инструментов.

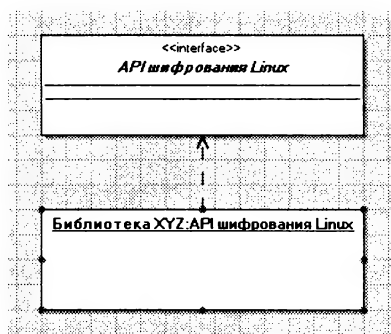


Рис. 8.13. Объект библиотеки XYZ является экземпляром класса API шифрования Linux

В прикладной программе объект практически всегда существует как экземпляр какого-либо класса или интерфейса. Для визуальной связи объекта с реализуемым им классом (интерфейсом) выбирается свойство *Instantiates* (Представляет) и в диалоговом окне выбирается класс или интерфейс, который будет реализован. После нажатия кнопки ОК на диаграмме автоматически сформируется соответствующая связь.

8.4.2. Диаграмма вариантов использования (Use case diagram)

Диаграммы вариантов использования (или диаграммы прецедентов) обычно применяют не на этапе разработки и не для создания конкретной модели элементов компьютерной программы, а раньше — на первых шагах общения с заказчиком. Они предназначены для построения достаточно общей абстрактной модели, понятной неспециалистам и описывающей, как будущая система должна работать с точки зрения пользователя, каково ее функциональное поведение, без погружения в детали реализации.

Актеры

Каждый вариант использования проектируемой системы отображает процесс взаимодействия с ней пользователя, выступающего в некоторой роли: Бухгалтера, Директора или Менеджера по продажам. Такие ролевые исполнители обычно называются Актерами. Отметим, что ими могут быть не только будущие пользователи, но и любые другие внешние объекты, взаимодействующие с системой в известной форме. При этом конкретные способы реализации такого взаимодействия не определяются, а рассматриваются только сценарии взаимодействия пользователей с проектируемой системой.

Диаграммы вариантов использования создают командой контекстного меню с пространства диаграммы **Add > Other Diagram > Use Case Diagram** (Добавить > Другая диаграмма > Диаграмма использование) — рис. 8.14.

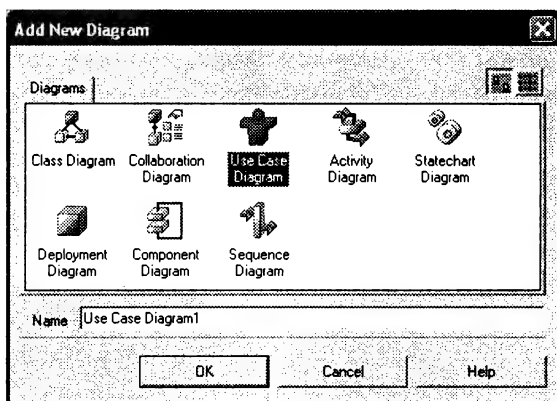


Рис. 8.14. Выбор диаграммы вариантов использования

Допустим, разрабатывается архитектура системы автоматизированного управления учебным процессом. Среди ее пользователей (актеров) можно выделить две большие группы: «преподаватели» и «студенты». Добавим их на диаграмму инструментом Actor (рис. 8.15).

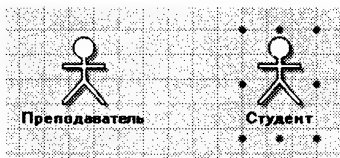


Рис. 8.15. Актеры на диаграмме вариантов использования

Соответствующие сущности в *UML* для наглядности представлены человечками.



Внутренняя структура актеров — внешних по отношению к проектируемой системе объектов — раскрытию не подлежит.

Варианты использования

Второй ключевой элемент для диаграмм данного типа — это вариант использования. Он определяет, какое действие должна выполнить система в ходе взаимодействия с конкретным актером. Это действие принято записывать в активной глагольной форме.

Добавляется вариант использования на диаграмму выбором элемента Use Case (Прецедент) на палитре инструментов. Этот элемент имеет форму эллипса, причем его внутренняя структура в рамках варианта использования не раскрывается. В то же время надо учитывать, что каждый вариант использования должен более-менее точно соответствовать некоторому функциональ-

ному блоку создаваемой программы, а не являться абстрактным комментарием или пожеланием. Вариант использования — это функция, которая должна быть обязательно реализована в будущем продукте. Поэтому общую внутреннюю структуру варианта использования (последовательность важных действий) можно указать добавлением к соответствующему объекту так называемых точек расширения из контекстного меню прецедента. Командой **Add ➤ Extension Point (Добавить ➤ Точку расширения)** можно вводить в прецедент список словесных команд, уточняющий действие.

В нашем примере добавим вариант **Поставить оценку** и уточним его двумя точками расширения (рис. 8.16). Как будет происходить взаимодействие между этими вариантами использования и актерами, будет показано далее. Реализуется оно с помощью отношений.

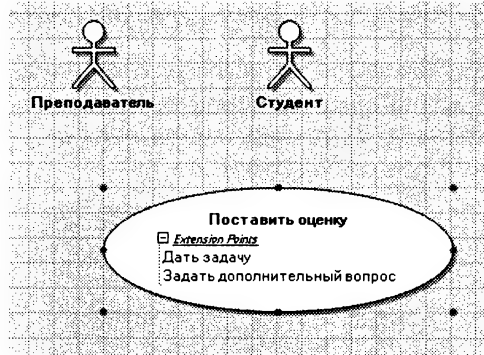


Рис. 8.16. Вариант использования с двумя точками расширения

В свойствах **Pre-condition** и **Post-condition** прецедента можно задавать действия (описывать условия), которые должны быть обязательно выполнены до запуска прецедента или перед его завершением.

Отношения между актерами и прецедентами

Отношения в UML возможны между актерами и вариантами использования, но не между вариантами использования друг с другом. Отношения делятся на четыре вида: ассоциации, обобщения, расширения и включения.

Отношения ассоциации. Этот наиболее общий вид отношений задает логическую связь между актером и прецедентом. На диаграмме оно изображается сплошной линией. Чаще всего отношение ассоциации используется для указания кратности отношений, что необходимо при формировании структуры базы данных.

В нашем случае можно сформировать связь между вариантом использования **Поставить оценку** и актером **Преподаватель**. Она формируется с помощью элемента **Communicates** палитры инструментов. При этом выставять оценки может как минимум один преподаватель, что можно явно указать на диа-

грамме. В Инспекторе объектов для выбранной связи кратность для актера фиксируется в свойстве Client Cardinality (в выпадающем списке можно указать значение «1»), а название конкретной актерской роли задается в свойстве Client Role (например, Прием зачета).

Две соответствующие характеристики для варианта использования задаются также в разделе Инспектора объектов для поставщика (Supplier). Для процесса постановки оценки кратность может быть задана в виде «1..*» — как правило, опытный преподаватель может одновременно оценивать знания сразу нескольких студентов (рис. 8.17).

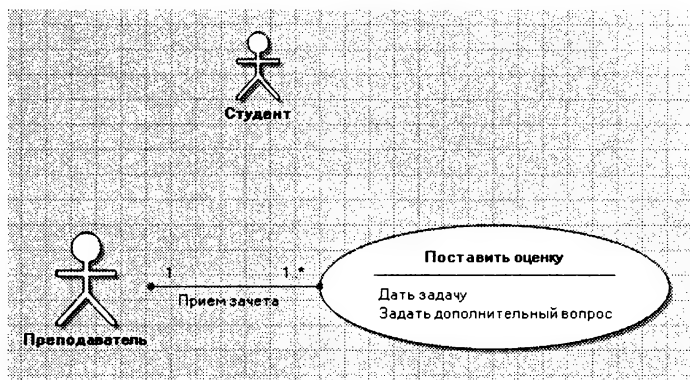


Рис. 8.17. Отношения ассоциации выражают базовое взаимодействие между актерами



Если кратность не указана, то она считается равной единице.

Отношение расширения. С помощью отношения расширения удастся выразить связь между двумя вариантами использования. Она задается пунктирной линией со стрелкой, направленной от расширяющего объекта к базовому, который дополняется, таким образом, новыми функциями. В программе отношение представляется условием, которое проверяется один раз при обращении к базовому варианту использования.

Например, перед тем как поставить оценку, преподаватель должен предварительно выполнить дополнительное действие — попросить студента предъявить зачетку (рис. 8.18).

Устанавливается эта связь с помощью инструмента Extends из палитры инструментов. Рядом с линией связи отображается ключевое слово <<extend>>.

Отношение обобщения. Это отношение указывает, что некоторый вариант использования (потомок) является частным случаем другого, более общего варианта (родителя). Связь обобщения (элемент Generalization палитры инстру-

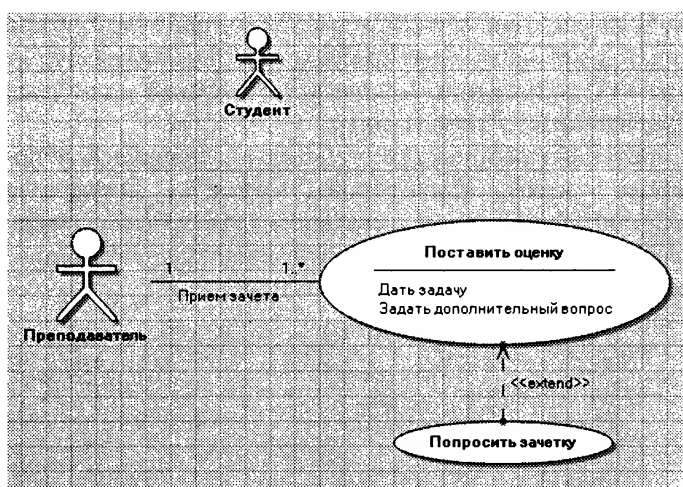


Рис. 8.18. Отношения расширения позволяют расширить функциональность взаимодействия между актерами

ментов) выглядит как сплошная линия с треугольной стрелкой, направленная от потомка к предку. Например, вариант использования Поставить оценку может быть частным случаем варианта Проверить знания (рис. 8.19).

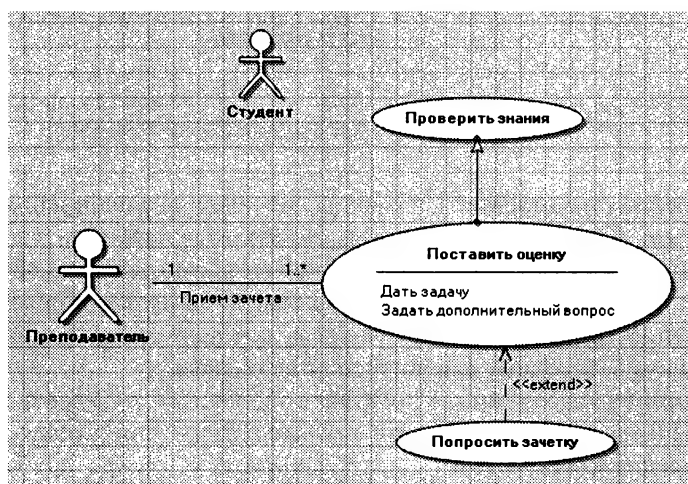


Рис. 8.19. Отношение обобщения на диаграмме вариантов использования

Отношение включения. Отношение включения указывает, что некоторый вариант использования является элементом, включенным в другой вариант использования. Оно задается инструментом `include` палитры инструментов, а изображается ключевым словом `<<include>>` и пунктирной линией со

стрелкой, направленной от охватывающего варианта к включаемому, входящему в него¹. Например, вариант использования Принять зачет включает в себя вариант Поставить оценку (рис. 8.20).

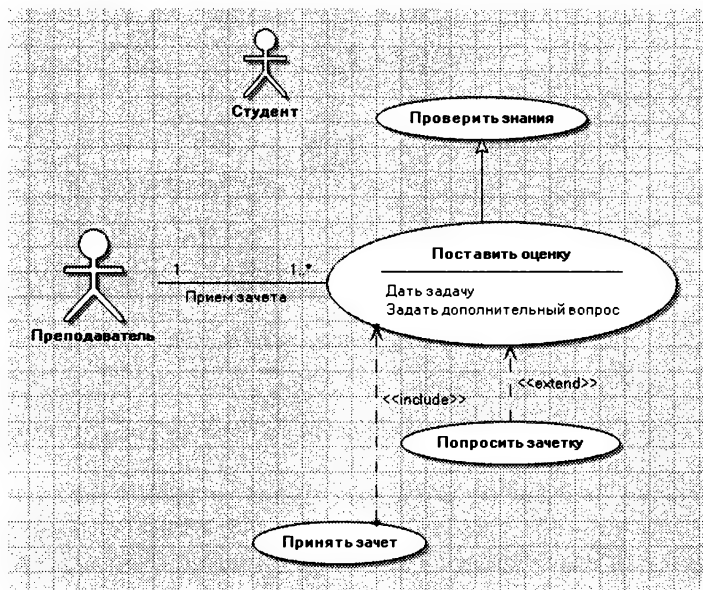


Рис. 8.20. Отношение включения на диаграмме вариантов использования

8.4.3. Диаграмма последовательности (Sequence diagram)

Рассмотренные выше виды диаграмм описывали лишь статические элементы системы, ее структуру и их взаимоотношения. Этого, конечно, недостаточно для построения более-менее точной модели. Поэтому в технологии UML имеется специальный вид диаграмм последовательности, позволяющий визуально представлять последовательности операций и описывать алгоритмы работы различных частей системы, явно зависящих от времени. При этом смысловая составляющая диаграмм этого вида охватывает лишь те объекты, которые взаимодействуют друг с другом по мере увеличения значе-

¹ Даже видные специалисты по использованию UML нередко противоречат друг другу и формальным рекомендациям стандарта, подчас задавая направления стрелок в связях расширения в противоположных направлениях. Иногда это направление меняется в одном и том же классе диаграмм при переходе от версии UML 1.5 к UML 2.0. Мы будем придерживаться одобренного большинством практикующих специалистов подхода, когда направление стрелок определяет лишь зависимость первого варианта использования от второго, но не последовательность выполнения (которая, как и любой временный фактор, в диаграммах прецедентов не учитывается).

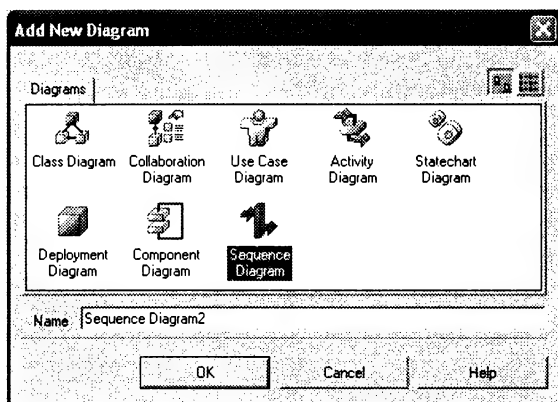


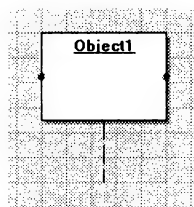
Рис. 8.21. Добавление диаграммы последовательности

ния некоторого таймера, и не учитывает взаимосвязи этих объектов с другими элементами системы.

Создание диаграммы последовательности выполняется командой контекстного меню с пространства диаграммы **Add ➤ Other Diagram ➤ Sequence Diagram** (Добавить ➤ Другая диаграмма ➤ Диаграмма последовательности) — рис. 8.21.

Объекты

Диаграмма последовательности представляется набором объектов, участвующих во взаимодействии. Объекты располагаются в ее верхней части в виде прямоугольников. Названия объектов подчеркнуты. Нередко через двоеточие после имени объекта указывается и имя класса. Нужный класс можно выбрать из доступных в рамках проекта с помощью свойства **Instantiates**. Объект размещается на диаграмме инструментом **Object** палитры инструментов.



Вниз от каждого объекта отходит пунктирная линия — своеобразная ось времени или «линия жизни». Она может заканчиваться символом X, обозначающим точку гибели объекта. Ниже этой точки на диаграмме может располагаться другой, новый объект, так как данный экземпляр считается полностью уничтоженным.



В качестве участников диаграммы последовательности наряду с объектами могут выступать и актеры (элемент **Actor**) — пользователи системы, проявляющие определенную активность. Таким способом удобно описывать взаимодействие человека с интерфейсными модулями. В свойстве **Stereotype** можно выбрать подходящий визуальный символ, обозначающий тот или иной тематический вид актера: от объекта управления (**control**) до базы данных (**database**).

Некоторые объекты могут существовать не с начала линии жизни до ее конца, а лишь на определенном интервале. В таких случаях верхний прямоугольник с названием объекта можно сдвинуть вниз к условной точке его возникновения.



Какой-либо конкретной шкалы на осях времени не ведется — отношения на них чисто принципиальные, показывающие, что случается раньше, а что — позже.

Обычно первым с левой стороны выступает объект, с которого стартует процесс визуализируемого взаимодействия. Сразу за ним указывается объект, активнее других общающийся с первым, далее — объект, активнее всех общающийся со вторым и так далее.

Важным понятием диаграмм последовательностей считается *фокус активности*. Он выделяется на линии жизни длинным узким прямоугольником, причем в процессе существования объекта фокус может как устанавливаться, так и исчезать.



С помощью фокуса в UML можно демонстрировать элементы рекурсивной работы объекта. Для этого в соответствующем месте фокуса активности, в правой его части, добавляется еще один тонкий прямоугольник.

Сообщения

Сообщения — это ключевые элементы диаграмм последовательностей. Именно с их помощью объекты взаимодействуют друг с другом. Сообщения обозначается линией (обычно горизонтальной) со стрелкой, которая направлена от объекта, посылающего сообщение (от клиента) к объекту-получателю (к серверу). Для визуализации отправки сообщений между объектами на палитре инструментов имеется пункт **Message** (Сообщения). Линия от клиента к серверу протягивается с помощью мыши (рис. 8.22).

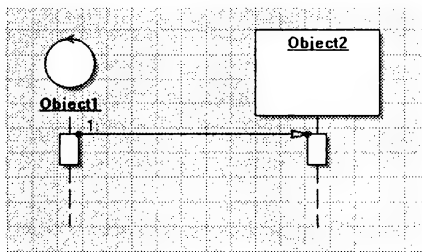


Рис. 8.22. Взаимодействие между объектами с помощью сообщений

Считается, что время передачи сообщения мало и его можно не учитывать, но все же если важно показать, что от момента отправки сообщения до его приема проходит некоторое время, можно воспользоваться формой свя-

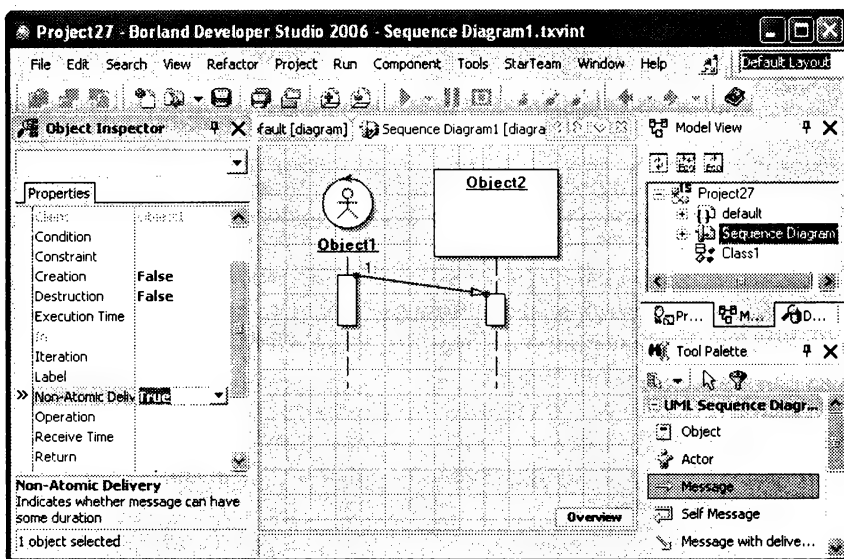


Рис. 8.23. Доставка сообщения с задержкой

зи Message with delivery time (Сообщение с задержкой). Объект-сервер в результате автоматически сдвигается вниз. Тот же эффект достигается с помощью свойства Non-Atomic Delivery (Распределенная доставка) сообщения, если оно равно True (рис. 8.23).

Как правило, смысл сообщения в диаграммах данного класса — извещение сервера о том, что ему необходимо выполнить некоторое действие. Свойство Synchronization (Синхронизация) позволяет уточнить форму сообщения. Оно может принимать одно из следующих значений.

- call. Обычное сообщение. Сплошная линия со стрелкой отходит от линии жизни клиента, а стрелка касается линии жизни сервера.
- asynchronous. Асинхронное сообщение (например, при исключительной ситуации) или промежуточное, показывающее очередной шаг процесса управления. Сплошная линия со стрелкой, имеющей один лепесток.
- synchronous, no wait. Синхронное сообщение, не требующее ожидания ответа от сервера, обозначается символом X или O рядом со стрелкой (рис. 8.24).
- wait. Сообщение, ответ на которое ожидается. Обозначается символом o рядом со стрелкой.



Сообщение может посылаться объектом самому себе, тогда стрелка показывается замкнутым кольцом. Этот элемент формируется на линии жизни инструментом Self Message.

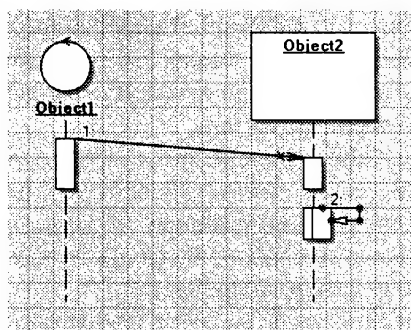


Рис. 8.24. Сообщение, не требующее ответа

Необходимо отметить удобную возможность той версии UML, которая реализована в Delphi, — возможность дополнения диаграммы последовательностей операторами языка. Это позволит, в частности, моделировать непосредственный процесс работы исходного кода. Для этого на палитре инструментов выбирается инструмент Statement Block (Оператор). После добавления блока на линию жизни будет предложено выбрать в диалоговом окне один из допустимых операторов языка Delphi (рис. 8.25).

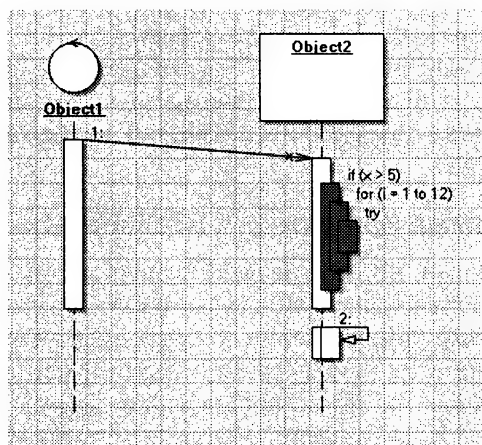


Рис. 8.25. Включение операторов языка Delphi в диаграмму UML

В итоге появляется возможность формировать многократно вложенные наглядные функциональные блоки.

Сообщения на диаграмме могут ветвиться в зависимости от некоторого условия. Если в зависимости от текущей ситуации сообщение от клиента может уходить к разным объектам, то соответствующее условие записывается рядом с каждой ветвью сообщения в свойстве Condition (Условие).

Для наглядности рядом с линией сообщения можно указывать одно из ключевых слов, поясняющих смысл этого сообщения. Так, допускается использование следующих стандартных идентификаторов.

- **call** — вызов серверного метода.
- **return** — возврат после вызова. Если выбрать инструмент **Return** палитры инструментов, то он автоматически соединит сервер, выполнявший действие, с клиентом, вызвавшим это действие. Таким образом указывается завершающее сообщение (пунктирная линия), информирующее о завершении некоторого процесса.
- **create** — требование создать объект. Оно может быть указано явно с помощью свойства сообщения **Creation** (Создание), равного **true**. В этом случае объект, принимающий сообщение, размещается сразу после стрелки, что обозначает его создание в соответствующий момент времени. Параметры процедуры создания со своими типами могут указываться в свойстве **Arguments** (Аргументы), а условия вызова — в свойстве **Condition** (Условие).
- **destroy** — требование уничтожить объект. Формируется заданием значения **true** свойству **Destruction** (Уничтожение).
- **send** — послать конкретный асинхронный сигнал, описанный в объекте-клиенте.

Комментарий к линии сообщения вводится в поле **Label** (Подпись). В некоторых случаях требуется указать операцию (свойство **Operation**), связанную с отсылкой сообщения, явно ввести время отправки сообщения (**Send Time**), время его доставки (**Execution Time**) и приема (**Receive Time**). При этом, хотя на диаграмме и будут показаны временные процессы, но, как уже говорилось, продолжительность тех или иных событий не указывается в абсолютных единицах времени.

8.4.4. Диаграмма кооперации (Collaboration Diagram)

Этот вид диаграмм UML охватывает те аспекты модели, которые не удастся представлять другими способами. В частности, диаграмма кооперации сочетает схематическое представление объектов модели вместе с их внутренней структурой (например, классы со свойствами и методами), с отображением динамических аспектов функционирования, правда, не связанных напрямую со временем. На ней задействуются те же элементы, что и в диаграммах последовательности, но линии жизни отсутствуют и возможные последовательности работ, шаги выполнения процессов, показываются на ней, как правило, с помощью порядковых номеров на линиях¹.

¹ Можно считать, что диаграммы кооперации демонстрируют некий образ, срез ситуации, возникшей во время функционирования модели на диаграмме последовательности. В ней все процессы как бы замерли, но известно, какой из них, за кем и в каком порядке будет выполняться. Поэтому между этими двумя видами диаграмм можно переходить командой контекстного меню диаграммы **Show as Sequence** (Показать последовательность) и наоборот — командой контекстного меню **Show as Collaboration** (Показать кооперацию). При этом, если альтернативная форма диаграммы отсутствует, она будет построена автоматически.

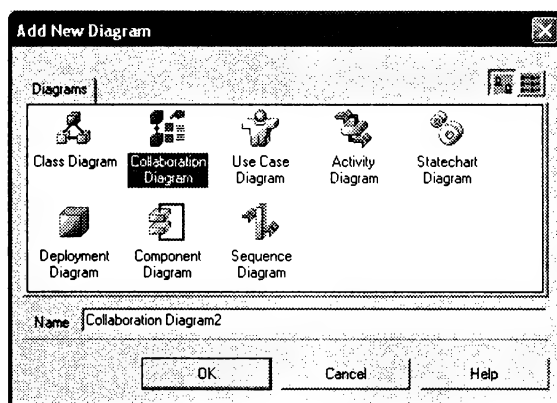


Рис. 8.26. Добавление диаграммы кооперации

Диаграмму кооперации создают командой контекстного меню с пространства диаграммы Add > Other Diagram > Collaboration Diagram (Добавить > Другую диаграмму > Диаграмму кооперации) — рис. 8.26.

Порядок шагов формируется на модели автоматически, после каждого нового добавления очередной связи. Рядом с соответствующей линией показывается стрелка и номер шага с двоеточием за ним. При этом нумерация шагов происходит гибко: система Delphi не позволит указать последовательность, в которой были бы смешаны синхронные и асинхронные сообщения. Последние просто выделяются дополнительным подпунктом в номере текущего сообщения.



Некоторый шаг последовательности не может быть выполнен, пока не выполнены все шаги, предшествующие ему по значению номера.

Объекты могут быть как одиночными экземплярами, так и их наборами. Для последнего случая надо присвоить значение true свойству MultipleInstance (Множественные экземпляры) — тогда соответствующий объект на диаграмме представится набором прямоугольников.

По умолчанию все объекты на диаграмме пассивны, то есть они не предназначены для явного управления другими объектами (что не исключает их способности к рассылке сообщений). Чтобы выделить на диаграмме активный объект, надо задать значение true его свойству Active. При этом прямоугольник активного объекта выделится широкой рамкой.

Рассмотрим пример диаграммы последовательности, в которой описывается обмен сообщениями между клиентом и Web-сервером (рис. 8.27).

Диаграмма кооперации может быть получена из нее автоматически, командой контекстного меню Show as Collaboration (рис. 8.28).

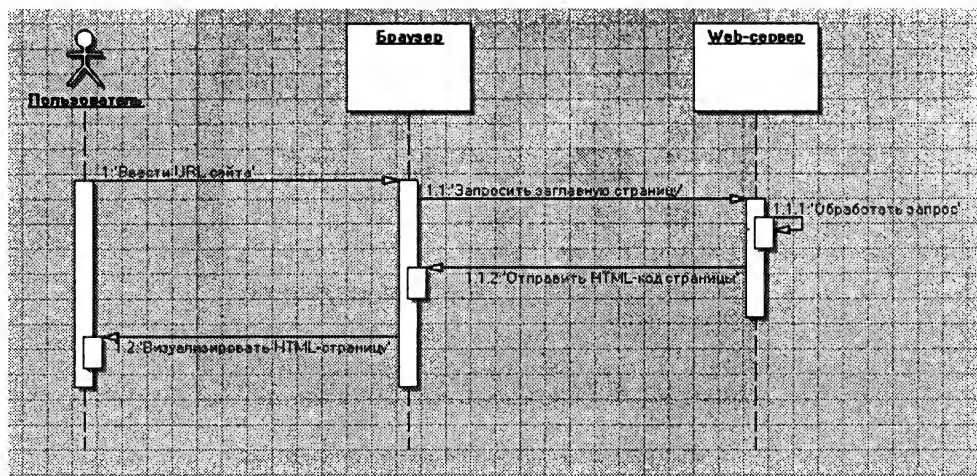


Рис. 8.27. Пример взаимодействия клиента с сервером на диаграмме последовательности

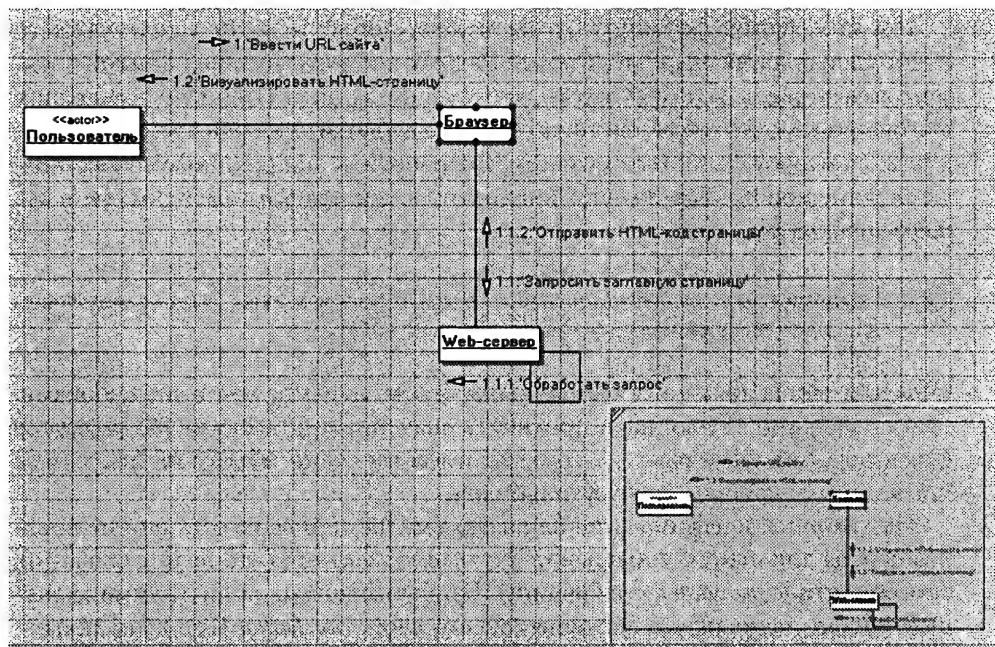


Рис. 8.28. Взаимодействие клиента с сервером на диаграмме кооперации

8.4.5. Диаграмма состояний (Statechart Diagram)

Диаграммы состояний, как и диаграммы последовательности, выражают не статическую структуру системы, а динамику ее поведения. Но если диаграммы, рассмотренные ранее, описывают последовательность взаимодействия различных объектов между собой, то данный класс диаграмм позволяет описать внутренние механизмы их работы, алгоритмическую логику. Только делается это не с помощью процедурных блок-схем, принятых в программировании, а с помощью автоматов или машин состояний.

Автоматы

Существует хорошо развитая теория, в которой изучаются вопросы построения и анализа работы автоматов. Ее базовые понятия: состояние и переход. Считается, что в определенный момент времени каждый автомат может находиться в одном из некоторого числа (возможно, бесконечного) допустимых состояний, и имеется набор правил, по которому допускается переход из одних множеств состояний в другие.

Эта модель сходна с моделью программирования тем, что переменные (ячейки памяти) в каждый такт процессора всегда хранят конкретные значения из допустимых диапазонов. А вычислительные операции, разрешенные над ними, вызывают переход этих переменных в новые состояния. Тот же алгоритмический принцип можно распространить и на большое число управленческих задач. В общем случае подход теории автоматов позволяет описать любую, сколь угодно сложную последовательность операций.

Рассмотрим базовые правила работы автоматов UML.

- Автомат может находиться только в одном состоянии. Хотя переменная может принимать множество значений, каждому из которых на диаграмме может соответствовать свой элемент, тем не менее в каждый момент времени (такт процессора) она хранит только одно из этих значений.
- Число состояний автомата конечно.
- Из текущего состояния автомат может перейти только в одно следующее состояние (не исключено, доступное на выбор из нескольких возможных).
- В рамках одной диаграммы у каждого состояния должен быть предшественник (за исключением начального состояния). Это правило запрещает размещение на одной диаграмме нескольких, никак не связанных друг с другом состояний.
- В любом из состояний автомат может находиться сколь угодно долго, а переходы между состояниями считаются мгновенными.
- На диаграмме состояний явно не фиксируется путь, по которому автомат перешел в текущее состояние (в отличие от диаграмм последовательности и кооперации). В переменной записано некоторое значение, и невозможно определить, в каких состояниях (значениях) она находилась ранее.

Состояния

Диаграмма состояний добавляется в проект командой контекстного меню с пространства диаграммы Add ➤ Other Diagram ➤ Statechart Diagram (Добавить ➤ Другая диаграмма ➤ Диаграмма состояний (рис. 8.29)

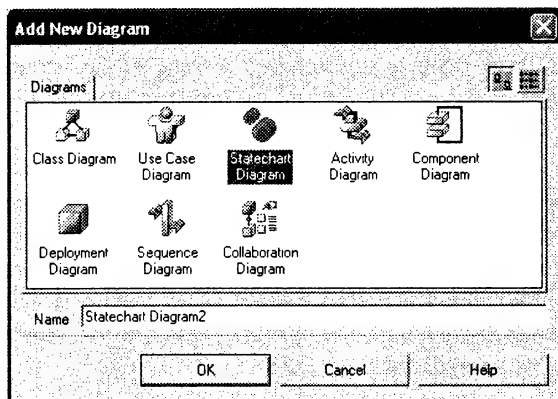


Рис. 8.29. Добавление диаграммы состояний

Конкретное состояние (в общем случае — «срез» некоторой системы) создается на диаграмме инструментом State с палитры инструментов. Состояние выражается прямоугольником с закругленными углами. Состояние имеет имя, которое записывается с прописной буквы, а под именем после черты записывается условие и действия. Условия могут быть проверены, а действия выполнены, когда автомат находится в данном состоянии.

На диаграмме также могут указываться начальное и конечное состояния автомата — элементы Start и Stop. Смена состояний отображается на диаграмме с помощью элемента перехода Transition (Переход). Переход ведется от первого, исходного состояния к последующему и выражается сплошной линией со стрелкой. Переход также может быть задан между одним и тем же состоянием. Выбрав некоторое состояние, укажем его же для операции Transition в качестве конечного — тогда линия перехода превратится в кольцо.

Некоторый переход может происходить в зависимости от определенного события (команды). Название этого события при необходимости указывается в свойстве Event Name (Имя события) перехода — тогда оно отображается рядом с соответствующей линией.

Для события может быть также задано сторожевое условие (логическое выражение), которое, если истинно, допускает выполнение перехода. Записывается оно в свойстве Guard Condition (Сторожевое условие), а на диаграмме показывается в квадратных скобках следом за именем события. Если это условие истинно, можно дополнительно задать некоторое действие, которое должно выполняться перед совершением перехода. Записывается это дей-

ствие в свойстве Action Expression (Выражение) и указывается на линии перехода через символ «/» следом за сторожевым условием.

На рисунке 8.30 показана диаграмма состояний банковского автомата, который последовательно переходит из состояния ожидания к вводу и проверке персонального идентификационного номера клиента (PIN) и далее к выдаче денег.

Отметим принятые правила оформления элементов этой диаграммы. Элементы State (Состояние) представляют собой не действие, а выражают некое статическое состояние автомата, в коем он может находиться неограниченно долго. Оно допускает перевод автомата в другие состояния при выполнении пользователем определенных действий или удовлетворении условий, указанных на линиях связи/перехода.

Подавтоматы

Внутри каждого состояния разрешается формировать собственный автомат – мини-диаграмму состояний. Это позволяет проектировать систему постепенно, раскрывая отдельные ее элементы по мере необходимости.

При формировании такого вложенного автомата (подавтомата) можно использовать инструмент Историческое состояние (History). Он позволяет временно сохранить текущее состояние подавтомата в случае прерывания работы (например, если возникла необходимость обработать какое-то внешнее или исключительное событие), а затем продолжить функционирование, не выполняя всех действий с самого начала, а стартовав сразу с отложенного состояния. Правда, запоминается состояние лишь в отношении текущего подавтомата, а вот если он, в свою очередь, состоит из других подавтоматов

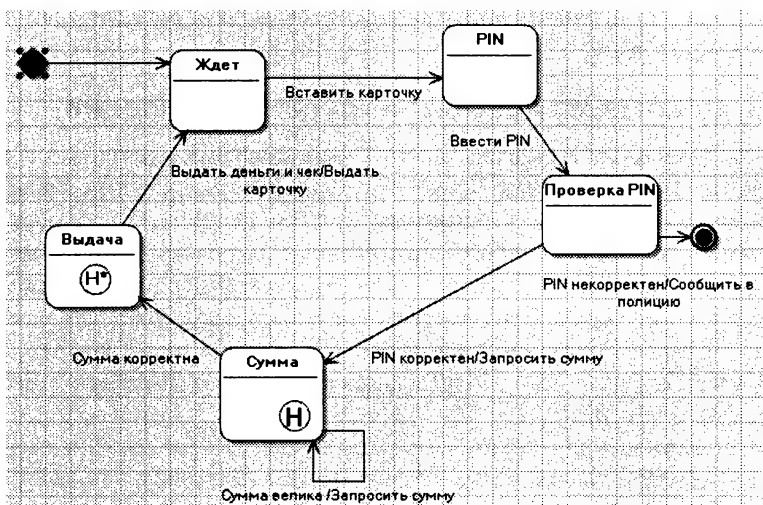


Рис. 8.30. Диаграмма состояний банкомата

и их состояние тоже необходимо сохранить, надо воспользоваться Глубоким историческим состоянием. Оно формируется из обычного Исторического состояния переводом свойства *Deep* в значение *True*. На диаграмме эти два вида исторических состояний отмечаются большой буквой *H* в кружке со звездочкой для Глубокого исторического состояния или без оной — для обычного (рис. 8.30).

От набора нескольких состояний автомата или подавтомата можно сразу перейти к другому внешнему состоянию, если для всех исходных состояний выполняется некоторое условие. Для изображения такой взаимосвязи применяется элемент *Fork/Join* (Ветвление/Соединение). Он бывает горизонтальным или вертикальным — *Vertical Fork/Join* и *Horizontal Fork/Join*. Отличие заключается лишь в ориентации линии, соединяющей несколько линий переходов в один.

После размещения элемента ветвления на диаграмме, с помощью связи *Transition* (Переход) сначала формируются переходы от исходных состояний по направлению к элементу соединения, а затем от него — к конечному. Другой вариант: от одного исходного состояния формируется промежуточный переход до элемента ветвления, и уже от него — к финальным состояниям. Последний случай означает начало параллельной работы нескольких подавтоматов, вложенных в одно состояние.



В редакторе UML Delphi не проверяется корректность соотношения числа и вложенности состояний.

Наряду с состояниями на диаграмме допускается размещение объектов — сущностей, которые делают выстроенную схему более наглядной. Элемент *Object* (Объект) представляется прямоугольником с подчеркнутым названием. Обычно его применяют для уточнения деталей работы автомата. Ведущих ролей он не играет.

8.4.6. Диаграмма деятельности (Activity Diagram)

Как частный случай диаграмм состояний диаграмма деятельности наиболее близка к общеизвестным блок-схемам программирования. Она дополняет элементы диаграммы состояний (состояния, переходы) набором элементов, необходимых для построения алгоритмических схем. Данный вид диаграмм нередко используют совместно с диаграммами классов, дополняя последние описанием логики работы методов классов. Диаграмма деятельности добавляется в проект командой контекстного меню с пространства диаграммы *Add > Other Diagram > Activity Diagram* (Добавить > Другая диаграмма > Диаграмма деятельности) — рис. 8.31.

В диаграммах деятельности введен новый графический элемент, схожий по внешнему виду с элементами состояний, — Деятельность (*Activity*). Он выражает некоторый алгоритм, выполняемый, когда до него доходит очередь. Визуальное отличие от диаграммы состояния заключается в том, что в состо-

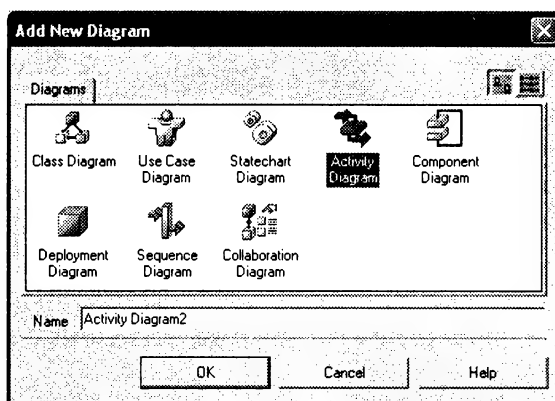


Рис. 8.31. Добавление диаграммы деятельности

янии присутствует черта, отделяющая название состояния, а у деятельности¹ она отсутствует.

Связи между деятельностями формируются с помощью элементов Transition (Переход). Различие переходов на диаграммах деятельности и состояний заключается в том, что первые считаются безусловными и выполняются обязательно. Дело в том, что в данном типе диаграмм существуют готовые средства визуализации условных действий, выделяемые особо, вне линий переходов. Хотя разработчик UML-диаграммы явно не ограничен возможностями Delphi и может формировать связи весьма произвольно, тем не менее желательно придерживаться общепринятых соглашений.

Для указания ветвления служит элемент Decision (Решение). Он представляется ромбом, под которым указывается условие ветвления. У ветвей указывается значение условия, при котором выполняется переход.

В ряде случаев полезно выделять в последовательностях действий смысловые зоны, например зоны ответственности между персоналом или потоки выполнения между процессорами. Для этого в языке UML существует графический элемент Дорожка (Swimlane), который позволяет визуально охватить некоторую последовательность действий.

Бывает, что деятельность активизируется заранее известными внешними событиями (сигналами). В дополнение к состояниям на диаграмме деятельности можно указывать отправителя и получателя сигнала с помощью элементов Signal Sending и Signal Receipt (рис. 8.32).

¹

Деятельность еще называют состоянием деятельности, что более точно выражает смысл элемента. Кроме того, возможно, более подходящее в данном контексте слово «действие» в русскоязычной литературе не применяется, так оно «зарезервировано» для диаграмм деятельности UML 2.0, где введено явное различие между деятельностью и составляющими ее более простыми действиями.

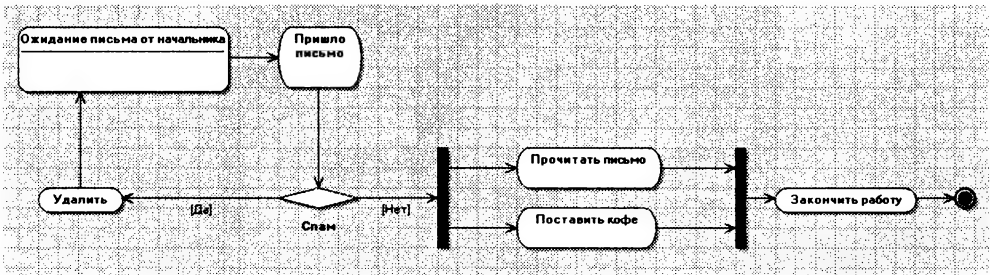


Рис. 8.32. Пример диаграммы деятельности с элементом состояния

На рисунке элемент Ожидание письма от начальника — не деятельность, а состояние (State), также доступное в рамках текущей диаграммы. А вот элементы деятельности определяются уже не названием некоторого статического состояния, а некоторой активностью, которую надо выполнить или на которую надо явно отреагировать для движения далее.

8.4.7. Диаграмма компонентов (Component Diagram)

В реальном проекте приходится учитывать не только абстрактные элементы модели, но и конкретные, физические — от файлов и сайтов до механических или электронных элементов создаваемой системы. Такие «реальные» элементы системы в технологии UML называются *компонентами* (component). Компонент представляет отдельный файл, элемент библиотеки или деталь механизма. Для создания компонентов служит диаграмма, добавляемая в проект командой контекстного меню с пространства диаграммы: Add ➤ Other Diagram ➤ Component Diagram (Добавить ➤ Другая диаграмма ➤ Диаграмма компонентов) — рис. 8.33.

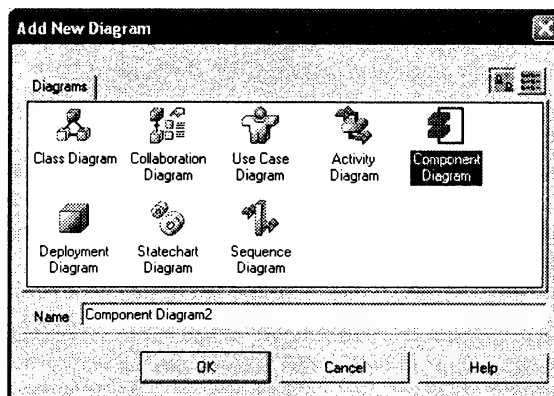


Рис. 8.33. Добавление диаграммы компонентов

Компонент (элемент **Component** на палитре инструментов) представляется прямоугольником, внутри которого имеются два декоративных раздела (в первых версиях технологии UML они предназначались для раздельного хранения данных и методов). Компонент имеет имя. Допускается вкладывание компонентов друг в друга.

В общем случае компонент может рассматриваться в модели как «черный ящик», поэтому для взаимодействия с ним требуется определить интерфейсы общения. Для этого задействуется соответствующий инструмент палитры инструментов **Interface**. Название интерфейса располагается вне его изображения. У одного компонента может быть несколько интерфейсов.

Для компоновки нескольких частей диаграммы в один модуль (пакет) задействуется элемент **Subsystem**.

Связи между элементами на диаграмме компонентов могут быть двух видов. Первая, **Supports** (Поддержка) предназначена для связи между компонентом и интерфейсом. Она не имеет направления и отображается сплошной линией. Подразумевается, что линия обозначает связь между интерфейсом и его реализацией в виде компонента.

Второй вид, **Dependency** (Зависимость) представляется пунктирной линией со стрелкой, направленной от элемента-пользователя к элементу, предоставляющему некоторую услугу (например, интерфейсу). Данный вид связи может формироваться для любых типов объектов диаграммы (за исключением случая, когда интерфейс сам выступает в роли клиента-источника) — рис. 8.34.

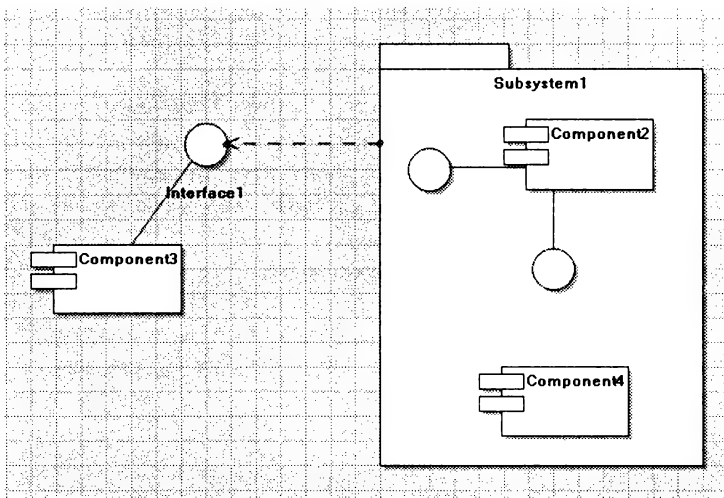


Рис. 8.34. Связи между интерфейсом и компонентами

8.4.8. Диаграмма развертывания (Deployment Diagram)

После того как система создана, ее предстоит внедрить на различных программных и аппаратных платформах. Особенности и способам такого внедрения и посвящен данный класс диаграмм. Диаграмма развертывания добавляется в проект командой контекстного меню с пространства диаграммы Add ➤ Other Diagram ➤ Deployment Diagram (Добавить ➤ Другая диаграмма ➤ Диаграмма развертывания) — рис. 8.35.

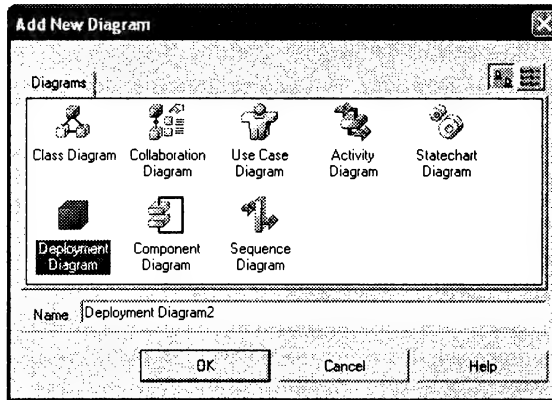


Рис. 8.35. Добавление диаграммы развертывания

Основным понятием диаграмм развертывания является *узел (node)* — в общем случае, некоторое устройство, способное к достаточно сложным действиям. В конкретных проектах в качестве узла обычно представляется компьютер, периферийное оборудование.

Узел на диаграмму помещается инструментом *Node (Узел)* палитры инструментов. Он показывается на диаграмме в виде трехмерного элемента. Если название узла не подчеркнуто, то оно трактуется как название класса, а если подчеркнуто, то как экземпляр класса (сам узел-класс указывается после названия экземпляра через двоеточие).

Внутри узла для наглядности можно размещать компоненты и объекты, а также интерфейсы. Между узлами, а также, в некоторых случаях, между входящими в них элементами, можно задавать связи-ассоциации (*Association*) с кратностями отношений, соединения зависимости (*Dependency*) и агрегации (*Aggregation*) — рис. 8.36.

8.4.9. Комментарий

Комментарий на диаграмме изображается прямоугольником с «загнутым» правым верхним углом. Текст комментария размещен внутри прямоугольника, а сам комментарий связывается с комментируемым элементом пунктирной стрелкой. Как правило, комментарий вводится в диаграмму инстру-

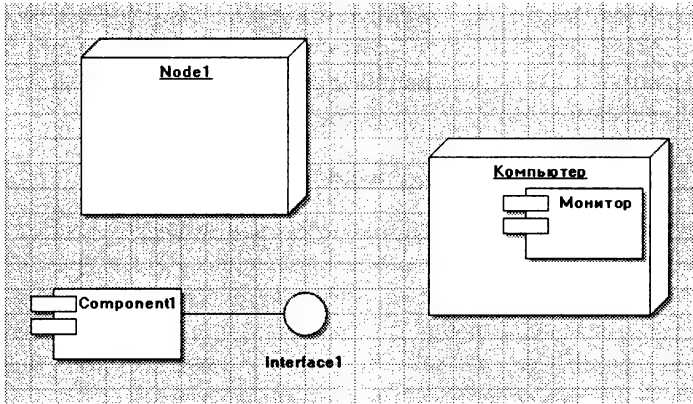


Рис. 8.36. Создание связей между узлами

ментом Note палитры инструментов, а связывается с комментируемым объектом посредством связи Note Link.

8.4.10. Экспорт диаграмм

Диаграммы UML можно экспортировать как готовые изображения. Это выполняется командой File > Export Diagram to Image (Файл > Экспортировать как изображение), после чего в диалоговом окне можно задать параметры результирующего изображения (вида модели) и предварительно его просмотреть.

Диаграмму также можно конвертировать в HTML-формат, для чего выполняется команда на построение автоматической документации. В кон-

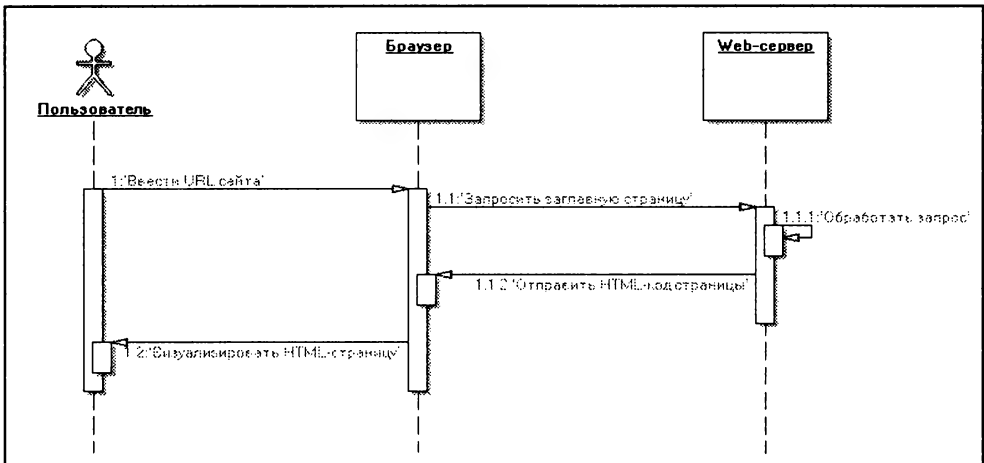


Рис. 8.37. Представление диаграммы, экспортированной в формат HTML

текстном меню диаграммы выбирается пункт **Generate Documentation** (Создать документацию) — рис. 8.37. При этом с помощью Java-апплета показывается также структура преобразованной диаграммы и предоставляется доступ к свойствам ее элементов.

8.4.11. Гиперсвязи (Hyperlinks)

Важнейшая особенность среды проектирования Delphi UML заключается в возможности построения диаграмм, вложенных друг в друга. Это осуществляется с помощью *гиперсвязей*, *гиперссылок* (*hyperlinks*). Они позволяют связать любой элемент любой диаграммы (или саму диаграмму) с любым элементом другой диаграммы. Названия элементов, дополненных гиперссылками, записываются не черным, а синим цветом.

Чтобы создать гиперссылку, надо выделить элемент на диаграмме (или саму диаграмму, щелкнув мышью на ее свободном месте), вызвать контекстное меню, выбрать пункт **Hyperlinks** ➤ **Edit** (Гиперссылка ➤ Правка) и в иерархической структуре диаграмм и объектов выбрать тот элемент, к которому будет вести данная ссылка (рис. 8.38).

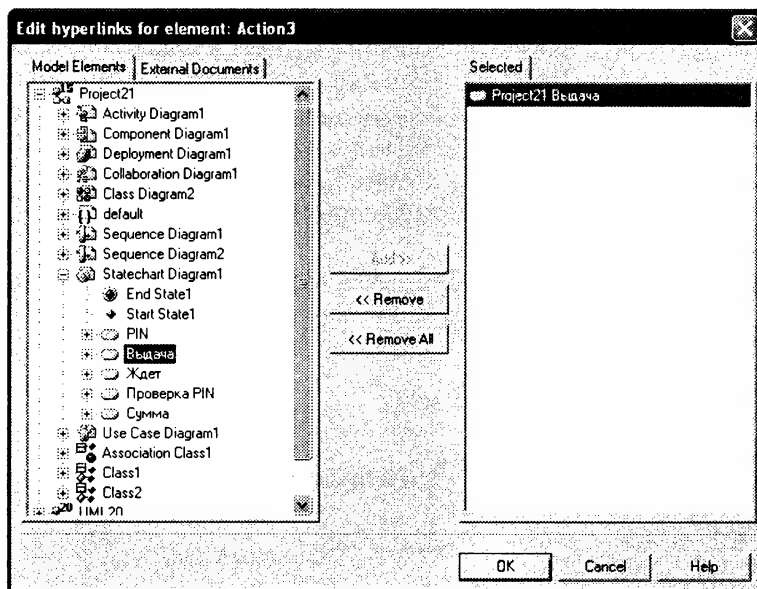


Рис. 8.38. Выбор пункта назначения для гиперссылки

В дальнейшем, если снова выбрать пункт **Hyperlinks** исходного элемента, то в нижней части его контекстного меню откроются новые пункты, выражающие возможные переходы к элементам других диаграмм (рис. 8.39).

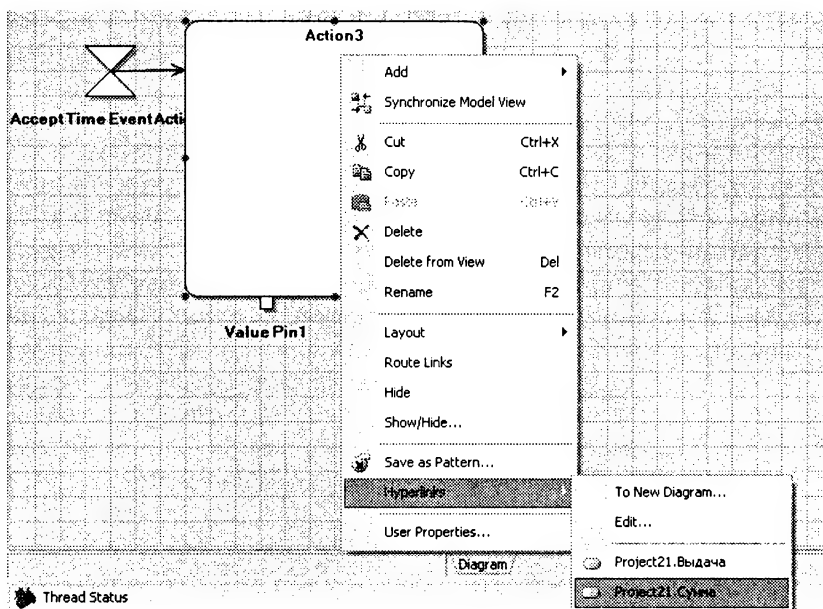


Рис. 8.43. Просмотр гиперссылок элемента

8.5. Технологии моделирования UML 2.0

8.5.1. Что нового в UML 2.0

Новая версия унифицированного языка моделирования UML 2.0 была официально принята консорциумом Object Management Group (OMG) в 2004 году. Эксперты OMG развивают язык UML в рамках стратегической концепции MDA. Она подразумевает разработку крупных приложений на основе модели. Поэтому улучшения языка UML коснулись прежде всего аспектов проектирования масштабных систем.

Всего в версии UML 2.0 насчитывается 13 типов диаграмм, из которых система Delphi поддерживает девять. Эти типы поделены на три группы:

- структурные диаграммы: развертывания (Deployment Diagram), классов (Class Diagram), компонентов (Component Diagram), внутренней структуры (Composite Structure Diagram), пакетов (Package Diagram, в Delphi отсутствует), объектов (Object Diagram, в Delphi отсутствует);
- диаграммы поведения: вариантов использования (Use Case Diagram), деятельности (Activity Diagram), машин состояний (State Machine Diagram);
- диаграммы взаимодействия, считающиеся подгруппой диаграмм поведения: последовательности (Sequence Diagram), взаимодействия

(Communication Diagram), временные (Timing Diagram, в Delphi отсутствует), обзора взаимодействия (Interaction Overview Diagram, в Delphi отсутствует).

Одним из наиболее мощных нововведений версии UML 2.0 стала концепция вложенных классификаторов. Любой элемент диаграммы UML (класс, объект, состояние) называется *классификатором*. Внутри любого классификатора можно располагать другие классификаторы, создавая, например, на диаграмме классы с машинами состояний внутри. Таким образом удобно реализуется один из базовых принципов проектирования программных систем «сверху вниз»: от большей абстракции к большей детализации.

Моделирование различной активности в версии UML 2.0 унифицировано. Теперь все модели UML, связанные с описанием действий, базируются на одном наборе определений. Любое действие, представленное на диаграмме UML, может быть параметризовано. Это упрощает его последующую реализацию на уровне программного кода в виде обычного метода с параметрами.

8.5.2. Диаграммы деятельности

Диаграммы деятельности изменились в версии UML 2.0 сильнее любых других диаграмм. В них появились новые элементы моделирования программной активности и средства контроля за этой активностью.

В диаграммах деятельности версии UML 2.0 введено понятие *деятельности (activity)*. Она складывается из более простых *действий (action)*. Действие выражается, как и в версии UML 1.5, небольшим прямоугольником со скругленными боковыми сторонами. Деятельность представляет собой укрупненный вариант такой фигуры.

Для элементов действия и деятельности можно указывать операции, которые должны быть выполнены предварительно. Они вводятся в свойствах Local precondition и Precondition элемента в Инспекторе объектов. Можно

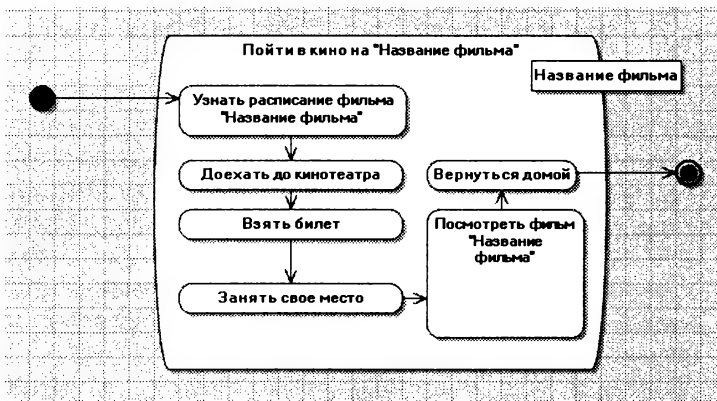


Рис. 8.40. Пример диаграммы деятельности с настраиваемым параметром

также задавать операции, вызываемые и по завершении работы. Для этого служат свойства *Local postcondition* и *Postcondition*. Операции записываются на языке OCL или в простом текстовом виде. Способ записи (OCL или текст) определяется свойствами *Precondition language* и *Postcondition language*.

Возможность параметризации элементов деятельности реализована в диаграммах деятельности UML 2.0 в виде элемента *Activity parameter*. Он представляет собой узкий прямоугольник с описанием параметра, размещенный в правом верхнем углу элемента *Activity*. Видимость параметра в рамках текущего пространства имен, как и видимость любого другого объекта, указывается в его свойстве *Visibility* в Инспекторе объектов.

8.5.3. Организация последовательных процессов

В диаграммах деятельности UML 2.0 выделяют два возможных вида организации последовательных процессов. Первый — это формирование потока управления. Второй — формирование потока данных.

Поток управления

Поток управления задает последовательность выполнения действий и деятельности на диаграмме. Он формируется с помощью элемента *Control Flow* палитры инструментов. Ход последовательности выражается на диаграмме деятельности темно-синей сплошной линией со стрелкой.

Финальная точка, в которой завершается поток управления в версии UML 2.0 разделена. Она представлена двумя разными элементами: финалом потока действия (*Flow Final*) и финалом деятельности (*Activity Final*). Если задействован финал потока действия, то, возможно, вся деятельность еще не завершена, а прекращен лишь один из ее потоков. В свою очередь финал деятельности полностью завершает функционирование деятельности.

Общий элемент логического ветвления диаграмм деятельности (*Decision*) разделен в версии UML 2.0 на два элемента. Один из них сохранил прежнее название: *Decision*. Другой получил название *Merge* (Слияние). Он способен принимать множество потоков управления. Однако на выходе у него может быть лишь один поток. В потоке управления используются также элементы слияния и объединения *Fork* и *Join*.



Элемент *Merge* передает управление далее, только если все входящие в него ветви управления были выполнены. В версии UML 1.5 для прохождения элемента слияния было достаточно выполнения любого из входных потоков.

В диаграммах деятельности версии UML 2.0 уточнена работа исполнительных элементов. Они выступают в качестве источников и приемников сообщений или сигналов, активизирующих заданную деятельность. Посылку сигнала активизации выражает элемент *Send Signal Action* палитры инструментов. Прием сигнала и начало работы при выполнении некоторого условия зада-

ются с помощью триггера — элемента Accept Event Action. Один или несколько подобных триггеров связываются с элементом, изображающим некоторое действие. Считается, что это действие начинает работу, только если сработают все заданные для него триггеры (произойдут назначенные им события).

В версии UML 2.0 существует особый вид триггеров — временной сигнал (элемент Accept Time Event Action). Он, как правило, задает некоторую точку времени на абсолютной или относительной шкале, достижение которой вызывает срабатывание дальнейшего действия. Фактически, этот элемент выражает процесс ожидания. Его образ символизируют песочные часы.

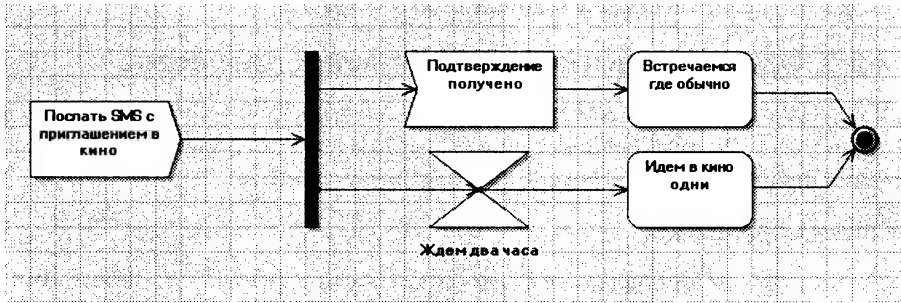


Рис. 8.41. Пример логического ветвления на диаграмме деятельности

Поток данных

Поток данных позволяет наглядно выразить процесс обработки данных. Его создают инструментом Object Flow. На диаграмме он представляется светло-синей сплошной линией со стрелкой. Поток данных может связывать действия только внутри конкретной деятельности.

В версии UML 2.0 применительно к потоку данных введены понятия входного и выходного разъемов. Они изображаются инструментами Input Pin

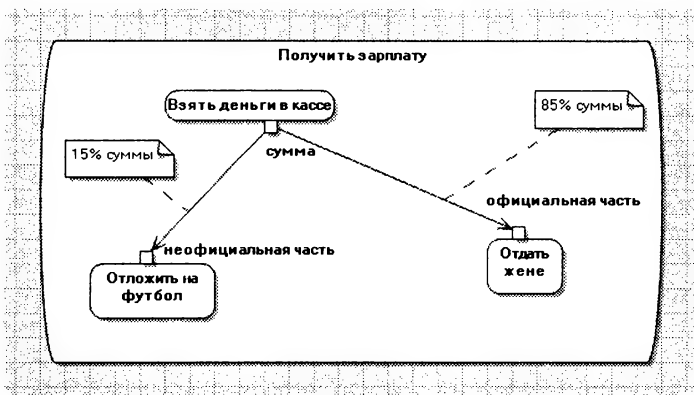


Рис. 8.42. Схема обмена данными на диаграмме деятельности

и Output Pin. В свою очередь элемент Value Pin отображает точку формирования некоторого значения.

Разъемы автоматически размещаются на границах элементов действий. Они связываются с другими внутренними составляющими деятельности и различными объектами с помощью потока данных (элемент Object Flow). Этот поток показывает не последовательность выполнения, а схему обмена информацией.

В диаграммах деятельности появились и другие новые элементы. Среди них: элемент Central Buffer (Центральный буфер), отображающий некоторое содержимое оперативной памяти, и элемент Data Store (Хранилище данных), отображающий содержимое жесткого диска и иных внешних носителей.

8.5.3. Диаграммы классов

В диаграммах классов версии UML 2.0 введено понятие порта и добавлен ряд новых видов связей. Порт уточняет форму отношений класса с внешним интерфейсом. Он выражает, является ли некоторый класс «потребителем» интерфейса (запрашивающим внешний интерфейс) или же его поставщиком. Порт также явно определяет кратность отношений между классом и его внешними пользователями («один к одному», «один ко многим» и т. п.).

Порт добавляется к существующему классу инструментом Port палитры инструментов. Его элемент располагается на одной из граней прямоугольника класса в виде маленького квадрата. Порт связывается с внешним интерфейсом одним из двух видов связей.

Первый вид связи порта с внешним интерфейсом называется *поставляемым интерфейсом (Provided Interface)*. Такая связь выражает предоставление классом некоторого интерфейса общего пользования. Наличие такой связи означает, что класс можно заменить связанным с ним интерфейсом. В свою очередь интерфейс реализуется классом, который скрыт от конечного пользователя.

Второй вид связи порта с внешним интерфейсом называется *запрошенным интерфейсом (Required Interface)*. Она выражает использование некоторым объектом внешнего интерфейса. С одной стороны линия связи соединяется с объектом. С другой стороны линия стыкуется с интерфейсом с помощью дуги окружности на ее конце.



Схемы стыковки классов и объектов с интерфейсами с помощью кругов и дуг окружностей иногда называют «леденцами на палочках».

Со стороны потребителей связь с интерфейсом желательно формировать не напрямую от класса, а через порт, связанный с классом. В результате удастся отделить описание класса от описания его связи с внешними интерфейсами. Это очень важная особенность версии UML 2.0, отчуждающая интерфейс от класса, реализующего данный интерфейс.



В языке UML интерфейсы традиционно представляются в виде кружков. Среда моделирования Delphi позволяет изменить вид интерфейса с прямоугольного, предлагаемого по умолчанию, на круглый. Для этого надо изменить свойство *Circle view* интерфейса в Инспекторе объектов. Когда оно равно *True*, интерфейс представлен в виде круга.

Интерфейсы присутствуют на разных типах диаграмм UML 2.0. В частности, они полезны при описании программных обращений к известным действиям системы. В таких случаях интерфейсы изображают «леденцами на палочках» — в виде небольших кружков. Название интерфейса наносят не внутри, а рядом с кружком. Если интерфейс связан с некоторым объектом пунктирной линией со стрелкой, то этот объект реализует только то, что явно требуется данным интерфейсом. Если же связь представлена обычной сплошной линией, то функциональные возможности объекта охватывают не только операции, указанные в интерфейсе, но и, возможно, некоторые другие.

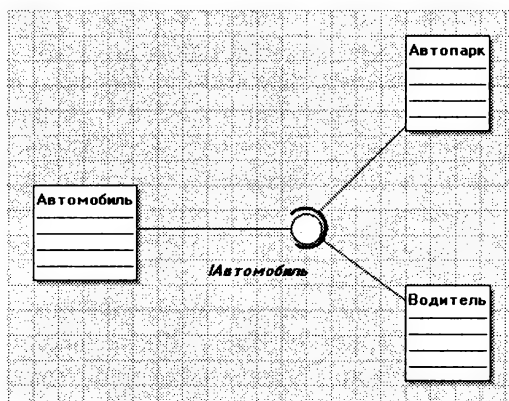


Рис. 8.43. Представление интерфейса на диаграмме классов

На диаграммах классов бывает необходимо представлять не только классы, но и описания их экземпляров. Сделать это позволяет новый элемент *Instance Specification*. Конкретный класс, который реализует этот элемент, определяется свойством *Instantiates*.

В версии языка UML 1.5 имелась возможность указания так называемых дискретных кратностей. Они позволяли представить допустимое количество объектов, участвующих в связи, в виде перечислимого набора значений (например «3, 5, 8, 11»). В версии UML 2.0 решено отказаться от такой мало востребованной возможности.

8.5.4. Диаграммы компонентов

Диаграммы компонентов в версии UML 2.0 претерпели весьма значительные изменения. Как и в диаграммах классов, в диаграммах компонентов появи-

лось понятие порта. Вместе с тем был удален элемент Подсистема (Subsystem). Добавилось понятие артефакта (Artifact) и класса (Class).

Артефакт в диаграммах компонентов обычно задействуется для представления двоичных файлов, документов, таблиц баз данных, сообщений электронной почты и других реально существующих объектов. Артефакт связан с конкретным файлом через свое свойство Filename. Класс связан со своей реализацией через свойство Implements.

В диаграммах компонентов UML 2.0 реализованы дополнительные виды связей, описывающие взаимодействие классов и интерфейсов. В частности, новый вид связи *Delegation Connector* позволяет соединить компонент с классом, который будет определять внутреннюю структуру компонента. Если компонент имеет некоторую реализацию (например, в виде физического элемента операционной системы — файла), то между компонентом и артефактом, задающим эту реализацию, прокладывается связь типа Realization.

На рисунке показан компонент Антивирус, который реализуется файлом ANTIVIR.EXE, структура которого описана в классе TAntivir.

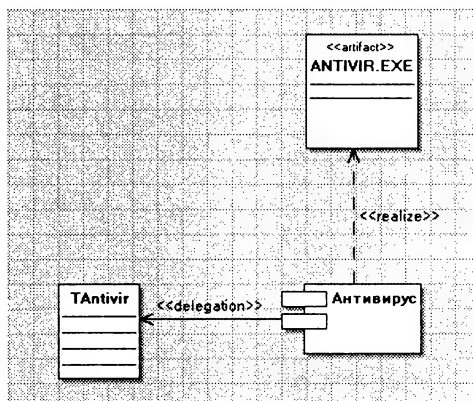


Рис. 8.44. Связь компонента с файлом реализации и типом представления

8.5.5. Диаграммы развертывания

В диаграммах развертывания версии UML 2.0 исчезло понятие компонента, существовавшее в версии UML 1.5. Оно заменено несколькими новыми элементами: *артефактом (Artifact)*, *устройством (Device)* и *средой исполнения (Execution Environment)*.

Артефакт диаграмм развертывания аналогичен артефакту из диаграмм компонентов. Устройство похоже на узел из диаграмм развертывания версии UML 1.5. Однако оно менее абстрактно и применяется для представления физических устройств. Элемент Execution Environment задействуется для представления операционного окружения. Элемент Interface из диаграмм развертывания UML 2.0 удален.

В диаграммах развертывания UML 2.0 появилась спецификация развертывания (элемент Deployment Specification). Она позволяет точно задать каталоги установки системы (свойства Deployment location и Execution location), а также местоположение некоторого файла (свойство Filename). Спецификация может дополняться собственными полями и операциями и допускает вложение артефактов и других спецификаций.



Артефакты могут предоставлять на всеобщее обозрение свои спецификации развертывания с помощью связи Manifestation.

Схема взаимодействия элементов диаграммы развертывания формируется с помощью коммуникационного пути. Он создается с помощью элемента Communication Path палитры инструментов и воспроизводится на диаграмме сплошной синей линией без стрелки (рис. 8.45).

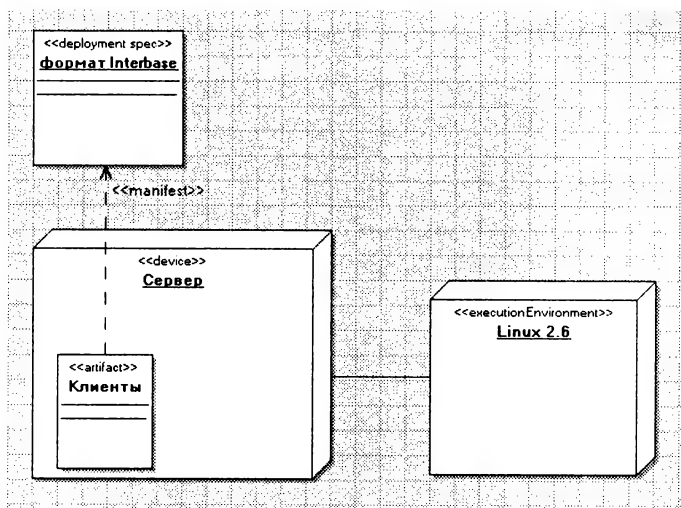


Рис. 8.45. Схема взаимодействия элементов на диаграмме развертывания

8.5.6. Диаграммы вариантов использования

В диаграммах вариантов использования UML 2.0 появился новый элемент Subject. Он дополняет прецеденты и актерские роли, что позволяет объединять тематические варианты использования в одно понятие.

Язык, на котором записывается условие начала или завершения работы прецедента, задается в его свойствах Precondition language и Postcondition language. Эти свойства могут принимать значение OCL (формальный язык объектных ограничений) или Text (текстовая свободная форма).

8.5.7. Диаграммы внутренней структуры (Composite Structure Diagram)

Диаграммы внутренней структуры появились в версии UML 2.0. Они позволяют описать внутреннюю структуру разных объектов. Такая структура может быть организована весьма сложно, иерархически, с многократным внутренним вложением элементов.



Диаграммы внутренней структуры называются иногда составными диаграммами.

В диаграммах внутренней структуры в дополнение к понятиям порта, класса и интерфейса введено понятие совместной деятельности (элемент Collaboration). В ее рамках разворачивается некоторое взаимодействие. Детали этого взаимодействия (так называемые случаи взаимодействия) представляются на диаграмме с помощью элемента Collaboration use. Он размещается внутри ранее созданной совместной деятельности.

Диаграммы внутренней структуры задают взаимоотношения между элементами некоторой системы за счет концепции ролей. Для описания ролей в совместной деятельности служит элемент Part. Роли с деятельностью связывают ролевой связью Collaboration Role, а случаи деятельности стыкуют с ролями посредством связи Role Binding.

Собственные связи между ролями формируют элементом Connector. Они бывают направленными и ненаправленными. Направленность связи задают ее свойством Directed. Вид связи конкретизируют ее свойством Type.

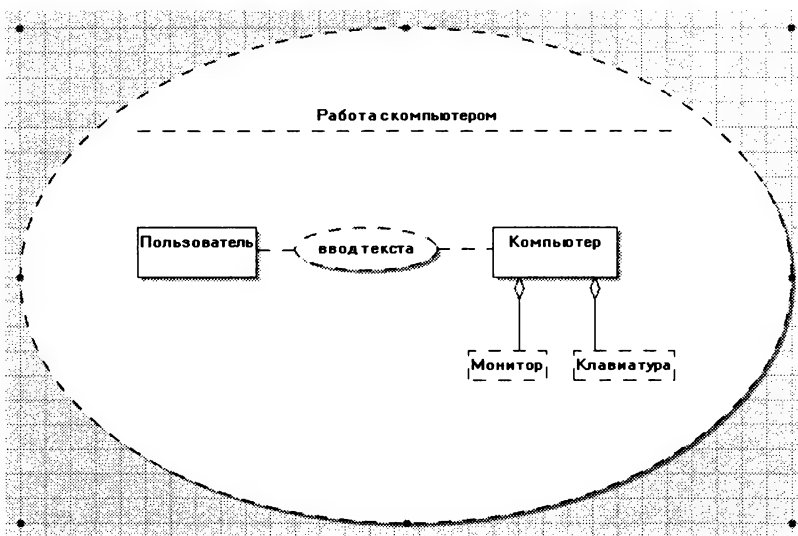


Рис. 8.46. Пример диаграммы внутренней структуры

8.5.8. Диаграммы последовательностей (Sequence Diagram)

Новый вид *диаграмм взаимодействия (Interactive Diagram)* в версии UML 2.0 состоит из двух дополнительных видов диаграмм: *диаграмм последовательностей (Sequence Diagram)* и *диаграмм коммуникаций (Communicate Diagram)*. Они представляют одну и ту же модель с разных точек зрения.

В диаграммах последовательностей версии UML 2.0 появился элемент *Lifeline (Линия жизни)*. Он заменяет ранее существовавший элемент *Object* и выражает как исходный объект, так и соответствующую ему временную линию существования.

Названия участников взаимодействия в диаграммах последовательностей можно задавать в более свободном виде, нежели в версии UML 1.5. Ранее использовалось понятие объекта, а название его подчеркивалось. В версии UML 2.0 введено понятие участника взаимодействия. Название его желательно записывать в формате «имя : класс».

В диаграммах последовательностей UML 2.0 улучшен способ выражения алгоритмических конструкций. В версии UML 1.5 для этого использовался элемент *Statement Block*. Теперь введено более общее понятие *фрейма взаимодействия*. Фрейм позволяет наглядно представлять охватывающие друг друга действия (блоки, циклы, условия).

К выделенной полосе активности на линии жизни объекта можно добавить составной, комбинированный фрагмент алгоритма (элемент *Combined fragment*). При его размещении на полосе активности среда Delphi предложит выбрать тип операции для создаваемого алгоритмического блока (например, loop — цикл). В результате на линии жизни сформируется прямоугольник с толстой каймой, в левом верхнем углу которого указано название фрагмента (например, loop). Список доступных типов операций приведен в таблице 8.1.

Таблица 8.1. Операции алгоритмических блоков

alt	Аналог условного оператора. Задаёт исполнение только того вложенного фрагмента, для которого указанное условие является истинным
opt	Вариант условного оператора, у которого отсутствует ветвь else
par	Параллельное выполнение всех вложенных фрагментов
loop	Цикл
region	Критическая область. Она должна быть выполнена за один шаг
neg	Выделение неверного фрагмента
ref	Ссылка на фрагмент, расположенный, возможно, на другой диаграмме
seq	Обозначение локальной диаграммы последовательности

Во фрагментах алгоритма на диаграммах последовательностей UML 2.0 можно указывать логику исполняемого действия на языке OCL. Для этого пунктиром выделяется вложенный в фрагмент блок. В его свойство *Constraint text* вводится выражение OCL. Если фрагмент подразумевает условное выпол-

нение, то таких пунктирных разделов в нем может быть два (по числу направлений ветвления).

В длину фрагмент увеличивается (по продолжительности) с помощью элемента *State invariant*. Выбрав этот инструмент на панели инструментов, следует щелкнуть мышью на линии жизни над областью удлиняемого фрагмента — и он увеличится. Подобные манипуляции бывают необходимы при согласовании некоторых событий на соседних линиях жизни.

Если в алгоритме надо сформировать ссылку на вынесенный блок или внешнюю диаграмму, следует использовать средство *Interaction use* палитры инструментов. Для охвата нескольких соседних линий жизни одним блоком служит инструмент *Tie frame*.

Элемент *Execution specification* служит для размещения на линии жизни точек, уточняющих конкретные моменты существования объекта. Этот элемент отображается в виде точки со стрелкой и присоединенным комментарием. Точки спецификации позволяют уточнить схему взаимодействия между объектом и интерфейсом. Сама точка соединяется с линией жизни объекта, видимого в текущем взаимодействии как интерфейс. А стрелка точки спецификации указывает на объект, которому данный интерфейс поставляется.

В версии UML 2.0 четко выделено различие между синхронными и асинхронными вызовами. Для отображения сообщений остался только один графический символ — *Message*. Но его внешний вид теперь можно задать с помощью свойства *Sort*. Синхронные вызовы отмечаются в этом свойстве значением *SynchCall*. Они выделяются на диаграмме сплошной линией с закра-

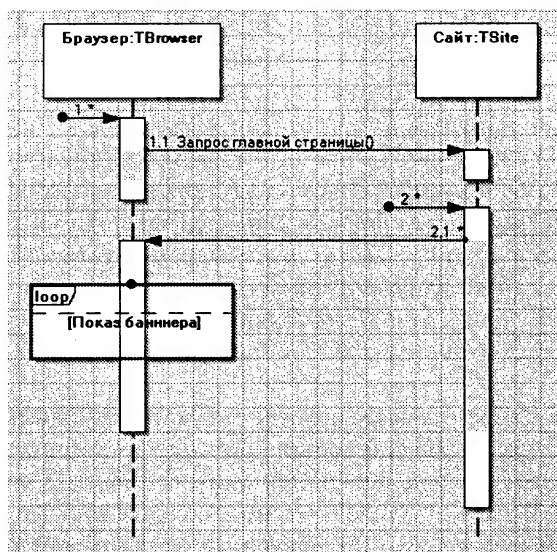


Рис. 8.47. Вызовы и сообщения

шенной стрелкой. Асинхронные вызовы задаются значениями `AsynchCall` (асинхронный вызов) или `AsynchSignal` (асинхронная отправка извещения) в свойстве `Sort`. Они выделяются серой открытой стрелкой.

8.5.9. Диаграммы коммуникации (Communication Diagram)

Диаграммы коммуникации в версии UML 1.5 назывались диаграммами кооперации. В версии UML 2.0 их существенно упростили, и теперь они позволяют показывать лишь объекты. Объекты теперь обозначаются элементом `Lifeline` (Линия жизни). Нумерация последовательности шагов выполняется, как и ранее, автоматически.

8.5.10. Диаграммы состояний (State Machine Diagram)

Диаграммы состояний стали новым типом диаграмм в версии UML 2.0. Они особенно важны для разработчика, использующего среду Delphi. Дело в том, что в последней версии Delphi 2006 имеется технология моделирования ЕСО III. Она расширена средствами визуального построения алгоритмов. С помощью этих средств описывается работа разных элементов модели. Ранее для описания модели и генерации исходного кода на языке Delphi применялись лишь статические диаграммы классов. Теперь задействованы и диаграммы состояний.

В основу диаграмм состояний положена концепция автоматов. Автомат представляет собой объект, который может находиться в одном из состояний ограниченного перечня и способен переходить из состояния в состояние по некоторым правилам. Доказано, что с помощью концепции автоматов можно запрограммировать алгоритм сколь угодно высокой сложности, если его можно записать на императивном языке программирования. Автоматы уже упоминались при описании диаграмм состояний UML 1.5.

Диаграмма состояний всегда относится к одному конкретному классу и описывает только его внутреннее функционирование. Конкретное состояние на диаграмме отображается с помощью элемента `State`. Начальное и конечное состояния (элементы `Initial` и `Final`) на диаграмме отображаются сплошным кружком и сплошным кружком с каймой, соответственно. Фактически, они представляют собой псевдосостояния, не возникающие в реальной работе. Начальное и конечное состояния лишь наглядно задают последовательность входа в первое рабочее состояние и выхода из последнего рабочего состояния.

Существуют версии начального и конечного состояний, которые являются локальными по отношению к конкретному состоянию. Они задают начало и конец последовательности действий уже внутри этого существующего состояния. В таком случае состояние, охватывающее другие состояния, называется *суперсостоянием*, а вложенные в него состояния — *подсостояниями*. Подобный подход иерархической организации состояний удобен тем, что позволяет сформировать общие (одинаковые) реакции различных под-

состояний на уровне одного суперсостояния. Допустим, имеется стандартное аварийное состояние и стандартный переход в него, вызываемый операцией отмены. Для каждого состояния на диаграмме можно индивидуально указывать этот переход, а можно объединить их в суперсостояние. Тогда переход в аварийное состояние указывается всего одной линией на диаграмме — она будет исходить только из суперсостояния.

Каждое из состояний на диаграмме состояний не обязано пассивно ожидать некоторого внешнего события, приводящего к переходу основного объекта в следующее состояние. Состояние может само выполнять некоторую внутреннюю деятельность (активность). Такая активность задается в свойстве *Do activity*. Это свойство состояния позволяет выбрать готовое действие из доступных в проекте (например, взяв с какой-нибудь из диаграмм деятельности). Подобное состояние, когда до него доходит очередь, запускает свое действие и ожидает его завершения, фактически отображая не статическое состояние, а выполнение процесса. Этот процесс может быть прерван, что вызовет переход в другое состояние. А если он закончится успешно, также выполнится соответствующий переход.

В диаграммах состояний UML 2.0, как и в версии UML 1.5, можно описывать историческое состояние. Только в версии UML 2.0 обычное историческое и глубокое историческое состояния разделены на два разных элемента: *Shallow History* и *Deep History*, соответственно.

В диаграммах состояний можно применять условный элемент *Choice* (Выбор). Другой элемент — *Junction* — выражает слияние и разделение потоков управления. Он представляет собой промежуточный узел, а по внешнему виду аналогичен начальному элементу *Initial*. В таком узле собираются, а потом вновь расходятся потоки управления.

Глава 9. ECO III : технология создания программ с помощью языка моделирования UML

9.1. Что такое ECO

Концепция *Enterprise Core Objects* (ECO, *ключевые корпоративные объекты*) представляет собой набор различных технологий и инструментов для построения масштабируемых систем на основе программируемой модели. В среде Delphi модель приложения представлена в виде диаграмм классов и диаграмм машин состояний UML. На основе модели выполняются основные шаги проектирования приложения ECO и его сопровождения.



В программной инженерии подход к созданию приложения на базе модели признан стратегическим. Он называется Model Driven Architecture (MDA, архитектура, управляемая моделью). Концепция MDA развивается консорциумом по объектному управлению OMG. В OMG, в частности, совершенствуется и язык UML.

Классический способ создания программ в среде Delphi состоит из двух шагов. На первом шаге проектируется пользовательский интерфейс в Дизайнере. На втором шаге в редакторе готовится программный код, который будет обрабатывать различные события пользовательского интерфейса. Недостаток этого подхода состоит в том, что по мере роста и усложнения проекта любые его модификации становятся все более и более трудоемкими, теряется возможность просмотра структуры всего приложения в целом. Не удастся визуально проверить корректность тех или иных взаимосвязей между классами, быстро проанализировать цепочку наследования, выполнить другие отладочные операции.

Технология ECO парирует эти проблемы. Почти весь процесс создания приложения решается манипулированием визуальной моделью. Она представлена в проекте Delphi в виде диаграмм UML. При этом удастся избежать ручной модификации исходного кода и минимизировать обращения к Дизайнеру пользовательского интерфейса. Весь исходный код приложения

генерируется из визуальной модели автоматически. Редактор и Дизайнер активно применяются лишь для тонкой настройки приложения согласно дополнительным требованиям пользователя.



Впервые технология ECO появилась в версии Delphi 8.

В системе Delphi 2006 реализована новая, третья версия технологии ECO. Она базируется на мощных возможностях среды моделирования Borland Together, специально встроенной в Delphi. В результате разработчик может управлять как статической структурой приложения с иерархией классов, так и поведением приложения, его программной логикой. Для этого задействуются так называемые диаграммы машин состояний UML, впервые появившиеся в версии UML 2.0. В итоге почти вся разработка сложного приложения может происходить в проектировщике диаграмм UML, встроенном в систему Delphi. На базе моделей, созданных таким образом, автоматически генерируется полнофункциональный исходный код приложения. Действует и обратная связь: когда исходные тексты модифицируются, структура модели автоматически подстраивается под внесенные изменения.



Двунаправленная технология синхронизации модели и кода получила в среде Delphi название LiveSource.

9.2. Модель ECO

Под моделью понимается описание некоторого реального или виртуального объекта, способного существовать физически или в воображении проектировщика. Модель может объединять несколько взглядов на объект, каждый из которых уточняет ту или иную особенность его существования. Модель компьютерной программы (в частности, модель ECO) отличается от физических и математических моделей большей полнотой, точностью и непротиворечивостью. На ее основе должен быть автоматически сгенерирован безошибочный и работоспособный исходный текст на заданном языке программирования. Для упрощения подготовки такой модели используются наборы объектов и дополнительные средства ECO. Они обеспечивают прозрачную взаимосвязь между программным кодом и элементами модели. В результате проектирование, развитие и сопровождение архитектуры всего приложения существенно упрощается.

9.3. Объектное пространство ECO (EcoSpace)

При запуске приложения ECO создается объектное (или модельное) пространство ECO. В программе оно называется *EcoSpace*. Содержимое объ-

ектного пространства образовано наборами экземпляров классов (объектов ECO), которые были описаны в диаграммах классов модели. В этом пространстве также хранится вся необходимая информация о модели. В пространстве ECO имеются механизмы создания и уничтожения объектов.

Среда времени выполнения ECO поддерживает автоматическое связывание содержимого пространства ECO с данными в файлах или базах данных. Для этого в нее встроены средства поддержки реляционной модели баз данных. Они автоматически преобразуют модель проекта в описание схемы базы данных (структуры и взаимосвязи таблиц). Это осуществляется автоматической генерацией сценариев SQL, которые занимаются физическим созданием и модификацией схем баз данных.

Для доступа к содержимому объектного пространства используются специальные компоненты — так называемые *дескрипторы ECO*. Они позволяют обращаться к пространству ECO из программного кода. С помощью дескрипторов программа выполняет необходимые манипуляции над *объектами ECO*.

Пользовательский интерфейс приложения связывается с содержимым объектного пространства через дескрипторы ECO. Элементы интерфейса снабжены специальными свойствами, через которые выполняется такая привязка. В результате просмотр и модификация объектов ECO реализуются простой настройкой свойств элементов интерфейса.

Удобнее всего применять технологию ECO для построения масштабных, корпоративных приложений. Она значительно сокращает время разработки приложений, в которых активно задействуются базы данных со множеством таблиц и сложными взаимосвязями, а также большое количество экранных форм. Создание подобных систем вручную, без предварительного построения модели, является трудоемкой рутинной задачей и обычно сопровождается множеством ошибок.

9.4. Этапы создания приложения ECO

Приложение ECO создается в следующем порядке.

1. Средствами UML формируется модель будущего приложения. Создаются классы, описываются их атрибуты и методы, настраиваются взаимосвязи между ними. На этом этапе возможно также дополнение классов машинами состояний. Они формализуют последовательности переходов различных элементов этих классов из одних состояний в другие.
2. В проект добавляются и настраиваются невизуальные компоненты (дескрипторы ECO). Они связывают созданную модель UML с прикладной частью проекта.
3. Проектируется пользовательский интерфейс. Задействуются компоненты, обеспечивающие связь интерфейса с моделью UML.

4. Создается переносимая логика приложения на языке объектных ограничений OCL. Элементы управления связываются с выражениями OCL для выполнения типичных стандартных действий (добавление, редактирование и удаление объектов ЕСО). Этот этап нередко выполняют с помощью визуальных средств Delphi. Большое число стандартных возможностей управления объектным пространством предлагается в виде готовых значений свойств элементов управления.
5. Настраивается связь пространства ЕСО с базой данных. В ней будет долговременно храниться его копия вместе со всеми объектами ЕСО.

По мере развития приложения этот цикл многократно повторяется. Принципиально важно, что все более-менее серьезные изменения, вносимые в проект, выполняются всегда над моделью, а не над исходным кодом и не над пользовательским интерфейсом.

Центральной концепцией ЕСО-программы считается ЕСО-пространство, в котором существуют различные ЕСО-объекты. Всевозможные аспекты поведения модели задаются на платформно-независимом языке объектных ограничений OCL.

9.5. Пример создания простого приложения ЕСО

9.5.1. Моделируем звездные системы

Для знакомства с возможностями технологии ЕСО рассмотрим простейший пример. Допустим, мы хотим организовать справочник по звездным системам. В нем должны храниться наборы таких систем: звезды и связанные с ними планеты с соответствующими названиями и характеристиками.



При желании вместо звезд и планет можно взять схему «начальник — подчиненные», «управление — отделы», «поставщик — продавцы», «сервис — клиенты» и так далее.

Для решения поставленной задачи потребуется смоделировать структуру условной звездной системы, состоящей из экземпляров таких классов, как Планета и Звезда. Конкретные планеты и звезды будут иметь как общие для всех свойства (название, размер, масса), так и индивидуальные характеристики. Например, для планет важной характеристикой является диаметр орбиты, а для звезд — температура. Каждый из экземпляров будет наследником класса «Элемент звездной системы». В этом классе сосредоточатся свойства, общие и для звезд, и для планет.

Следуя обычной схеме разработки, программист приступил бы к созданию классов ручным кодированием. Сначала он подготовил бы новый модуль, в его интерфейсном разделе описал бы соответствующие классы, отследил схему наследования и реализовал нужные методы классов. Затем он присту-

пил бы к разработке пользовательского интерфейса и процедуры заполнения таблиц. В эти таблицы необходимо записывать значения полей экземпляров разных классов, отслеживать их изменения, хранить массивы всех объектов, сохранять их содержимое на диске и выполнять другие операции. При этом разработчику пришлось бы мыслить преимущественно в терминах конкретных операторов языка, команд описания типов и классов. А поскольку общая структура приложения формально не описана, такая работа чревата постоянными ошибками. Например, функциональные возможности классов могут противоречить друг другу, а могут, наоборот, дублироваться. Запомнить же все взаимосвязи и зависимости между классами в более-менее крупном проекте практически невозможно.

Сейчас мы покажем, сколь сильно отличается этот традиционный и морально устаревающий подход от высокоуровневой технологии моделирования ECO.

9.5.2. Создаем первый абстрактный класс

Реализуем вышеописанную простую взаимосвязь между понятиями «Элемент звездной системы» — «Звезда» — «Планета» с помощью технологии ECO. Дадим команду **File** ➤ **New** ➤ **Other** (Файл ➤ Создать ➤ Другое) и выберем значок **ECO WinForms Application** (Приложение ECO Windows Forms) — рис. 9.1.

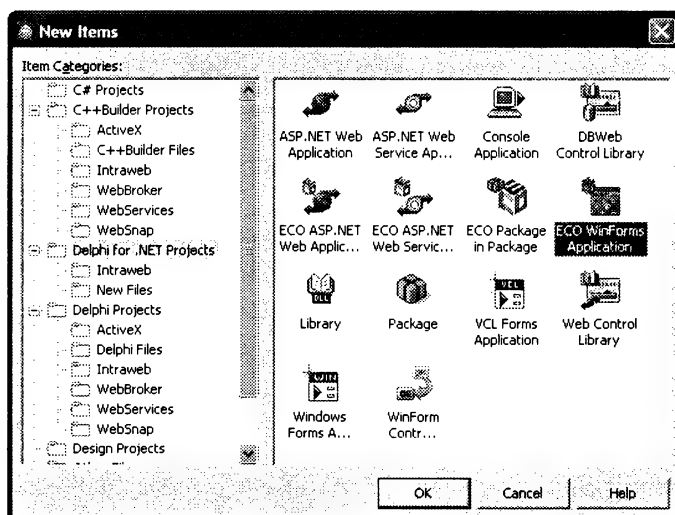


Рис. 9.1. Создание приложения ECO

В следующем окне укажем название проекта (например, `ECO1Project`) и его местоположение (папку). После создания пустой заготовки откроется окно Дизайнера, в котором представлена начальная форма приложения. Посмотрим на структуру автоматически созданной модели (пустой заготов-

ки) командой View ➤ Model View (Просмотр ➤ Модель). Эта команда вызывает средство просмотра модели (рис. 9.2).

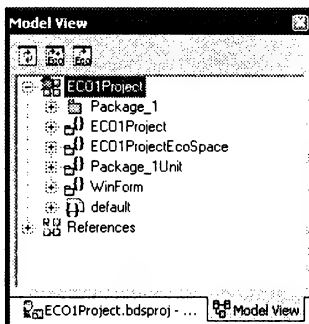


Рис. 9.2. Средство просмотра модели

Кроме самого проекта (ECO1Project) и главной формы (WinForm) в модели присутствуют новые элементы, добавленные в нее автоматически. Среди них отметим следующие.

- ECO1ProjectEcoSpace — объектное пространство ECO. Это основное хранилище, в котором будут располагаться экземпляры создаваемых классов модели во время работы программы. Пространство ECO также отвечает за связь с внешним источником данных, например за связь с базой данных или файлом XML.
- Package_1 — пакет классов UML, которые мы будем создавать.



Исходное окно модели проекта может быть пустым. Чтобы в нем отразилась текущая структура, проект необходимо полностью сохранить, выполнить компиляцию и в средстве просмотра модели нажать кнопку Refresh Model View (Обновить).

Создание модели начинаем с перехода в режим моделирования. Для этого дважды щелкнем мышью на строке Package_1, и среда Delphi откроет окно построения диаграмм классов UML. Его центральная часть, напоминающая окно Дизайнера, называется поверхностью моделирования (рис. 9.3).

Разместим на форме первую диаграмму. Для этого выберем на палитре инструментов в категории UML ECO Class Diagram инструмент ECO Class. Щелкнем мышью в подходящем месте на поверхности моделирования — будет создана диаграмма классов.



Описание диаграмм классов приводилось в главе, посвященной языку UML.

Введем название созданной диаграммы: StarSystemElement («Элемент звездной системы»). Пробелы в названии класса не допускаются. Символы русского алфавита использовать не рекомендуется, хотя это и возможно (рис. 9.4).

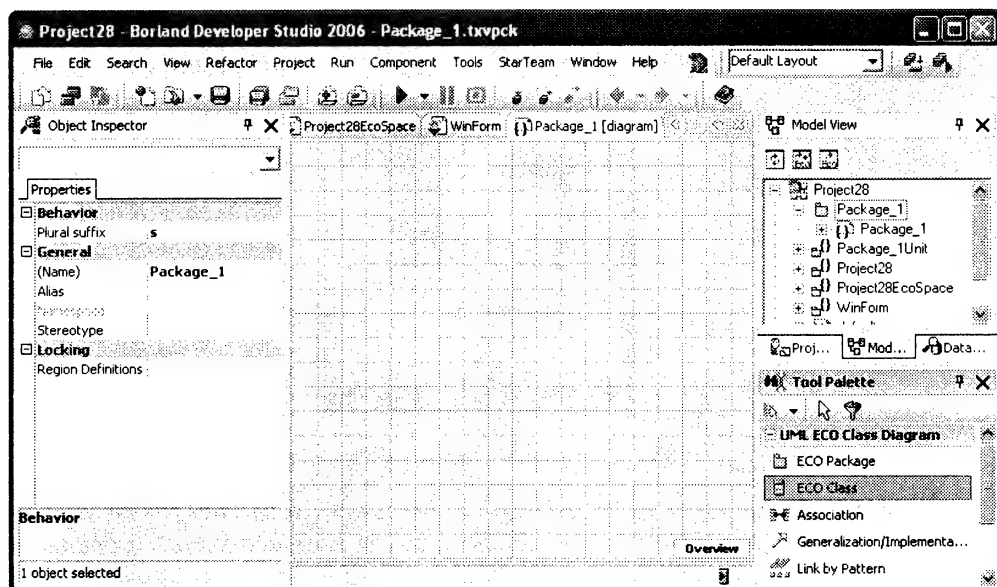


Рис. 9.3. Поверхность моделирования

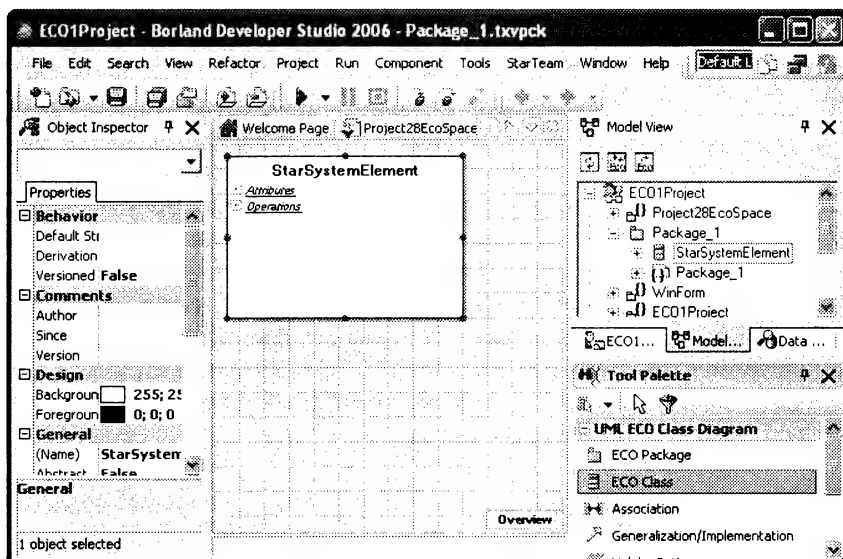


Рис. 9.4. Создаем класс StarSystemElement



Диаграмму класса можно добавить через контекстное меню поверхности моделирования командой **Add > Eco Class** (Добавить > Класс) или нажатием комбинации клавиш **Ctrl + L**.

Полученная диаграмма визуально представляет структуру класса `StarSystemElement`. На данном этапе проектирования надо решить, нужны ли в нашей программе абстрактные классы (классы, экземпляры которых создавать запрещено). Исходя из условий задачи, нам потребуются экземпляры будущих классов «Звезда» и «Планета». Они станут наследниками класса `StarSystemElement`. А вот сам «Элемент звездной системы», судя по всему, не может быть использован для создания объектов, поэтому будем считать его абстрактным классом. Чтобы отметить это, выделим на поверхности моделирования единственный пока класс `StarSystemElement` и в Инспекторе объектов назначим для него значение `True` в свойстве `Abstract`. В результате шрифт в названии этого класса на диаграмме изменится и станет наклонным. Это обозначает абстрактность класса.

9.5.3. Строим иерархию классов

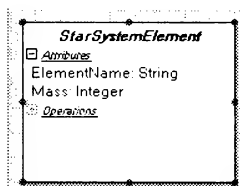
Определим свойства, общие для всех трех классов модели. Пусть первым таким свойством станет название объекта — `ElementName`. Его не надо путать со свойством `Name`, которое в исходном тексте программы используется в качестве названия идентификатора класса. Новое свойство `ElementName` определяет название элемента звездной системы (например, «Солнце», «Земля», «Альфа Центавра»). Вторым общим свойством станет масса звезды или планеты в некоторых условных единицах — `Mass`.

Новые свойства добавляются к классу обращением к команде контекстного меню его элемента на диаграмме: **Add > Attribute** (Добавить > Атрибут). Вводим название нового свойства: `ElementName`, выделяем его и в Инспекторе объектов, в поле `Type` (тип свойства) вручную введем строку `"string"` (символьная строка). Аналогичным способом добавим свойство `Mass`. Пусть его тип останется по умолчанию `"integer"` (целое).

Теперь диаграмма класса `StarSystemElement` выглядит следующим образом.

Добавим на диаграмму два новых класса: `Star` (Звезда) и `Planet` (Планета). Введем в класс `Star` дополнительное свойство `T: Integer` (температура), а в класс `Planet` — свойство `Orbit: Integer` (орбита). Кроме того, следует выключить свойства `Abstract` этих классов, ведь на их основе в программе будут созданы экземпляры объектов.

Теперь необходимо указать, что классы `Star` и `Planet` являются наследниками абстрактного класса `StarSystemElement`. Выполним это с помощью средства `Generalization/Implementation` с палитры инструментов. Выбрав его, щелкнем на элементе диаграммы `Star` — курсор превратится в перечеркнутый кружок.



Наведем его на элемент класса `StarSystemElement`, свойства которого наследуются, — курсор примет вид небольшого перекрестья. Выполним здесь второй, финальный щелчок. Между двумя элементами протянется направленная связь. Аналогично установим связь и между элементами `Planet` и `StarSystemElement`.

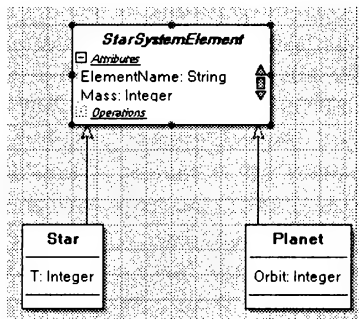


Рис. 9.5. Связь между классами

По мере построения иерархической структуры наши действия отражаются в Просмотрщике модели. В нем фиксируется появление новых классов.



Важно периодически компилировать весь проект. Технология ECO использует стандартное пространство имен `System.Reflection.NET` для анализа структуры классов, загруженных в модель, их типов данных, полей и методов. Поэтому проект (сборка .NET) должен быть преобразован в законченный двоичный вид, чтобы его структура корректно отразилась в Просмотрщике моделей.

Сохраним и откомпилируем проект. Ошибок в нем быть не должно — описания всех классов будут сгенерированы автоматически.

9.5.4. Добавляем дескрипторы ECO

Связь программного кода и пользовательского интерфейса с моделями ECO обычно происходит через компонент `ExpressionHandle`. Он доступен в категории `Enterprise Core Objects` палитры инструментов. Через подобные идентификаторы ECO организуется доступ к объектному пространству.



Идентификаторы объектов ECO часто называются дескрипторами ECO.

Программист связывает идентификаторы ECO друг с другом в цепочки. Это позволяет обращаться к отдельным областям объектного пространства и целенаправленно их обрабатывать.

Во главе цепочки идентификаторов должен быть расположен *корневой* (*root*) идентификатор. Он задает общий контекст работы для всех остальных идентификаторов проекта. Корневой идентификатор обычно добавляется к

проекту автоматически, а прочие идентификаторы должны быть его прямыми или косвенными наследниками. Они предоставляют доступ к конкретным наборам объектов пространства ЕСО.

Переключимся в режим проектирования формы. В нижней части окна Дизайнера уже имеется набор различных компонентов ЕСО. Они поддерживают базовые функциональные возможности пустого приложения. Добавим к проекту компонент `ExpressionHandle` и назовем его `ehStar` (свойство `Name`). В свойстве `RootHandle` (Корневой идентификатор) выберем значение `rhRoot`. Это значение является ссылкой на автоматически созданный корневой идентификатор. Таким образом, мы задали место дескриптора `ehStar` в цепочке идентификаторов текущего проекта ЕСО. Дескриптор `ehStar` является прямым наследником корневого идентификатора.

Действия, которые будут выполняться с помощью данного дескриптора, определяются содержанием его свойства `Expression`. В нем следует записать выражение на языке объектных ограничений OCL. Это выражение определяет механизм создания объектов нашей модели во время работы программы.

Мы будем использовать дескриптор `ehStar` для создания экземпляров класса `Star`. Поэтому в свойстве `Expression` объекта `ehStar` введем строку

```
Star.AllInstances.
```



Язык OCL в данной книге не рассматривается. С ним следует ознакомиться по учебникам, посвященным языку UML и программному моделированию.

9.5.5. Проектируем пользовательский интерфейс

С помощью дескриптора `ehStar` мы организовали первую связь объектного пространства с клиентской частью приложения. Теперь можно перейти к проектированию пользовательского интерфейса.

Допустим, мы хотим сформировать таблицу, представляющую созданные нами экземпляры класса `Star`. В таблицу они вводятся кнопкой **Добавить звезду**. Необходимую работу мы проведем исключительно с помощью визуальных средств Delphi и технологии ЕСО, не прибегая к ручному программированию.

Разместим таблицу на форме с помощью инструмента `DataGrid` из категории `Data Controls` палитры инструментов. Исходно таблица пуста. В ее свойстве `CaptionText` (Заголовок таблицы) введем строку **Звезды**, а на форму поместим кнопку **Добавить звезду**.

Теперь установим связь между дескриптором `ehStar` (поставщиком экземпляров класса `Star`) и созданной таблицей. Для этого в свойстве `DataSource` таблицы, в списке доступных идентификаторов, выберем значение `ehStar` — тогда в таблице появятся поля класса `Star` и его родительского класса `StarSystemElement` (рис. 9.6).

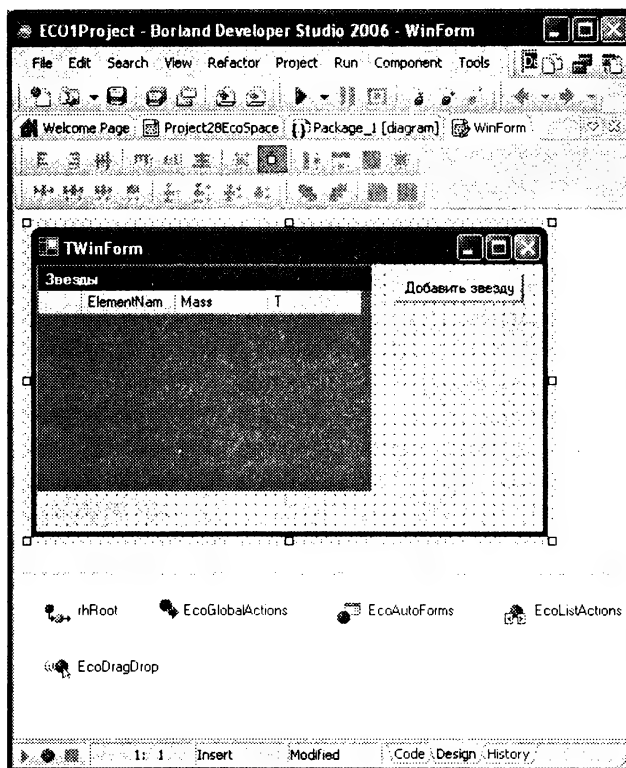


Рис. 9.6. Таблица звездных систем

Если теперь запустить программу, то мы увидим одну пустую таблицу. Однако заголовки ее столбцов соответствуют названиям как собственных полей класса *Star*, так и полей, унаследованных от родительского абстрактного класса *StarSystemElement*.

9.5.6. Настраиваем элементы интерфейса

Настроим ранее созданную командную кнопку на добавление в таблицу новых объектов. Выделим в Дизайнере эту кнопку. В ее свойстве *EcoListAction* (Список стандартных действий ECO) выберем значение *Add* (Добавить объект), а в свойстве *RootHandle* выберем нужный нам дескриптор *ehStar*. Он конкретизирует действие метода *Add*: создает в объектном пространстве экземпляр класса *Star*.

Созданный экземпляр надо связать с визуальным элементом, который будет его хранить и представлять на экране (в нашем случае это таблица *DataGrid1*). Для этого в свойстве кнопки *BindingContext* (Контекст связывания) выберем значение *DataGrid1*.

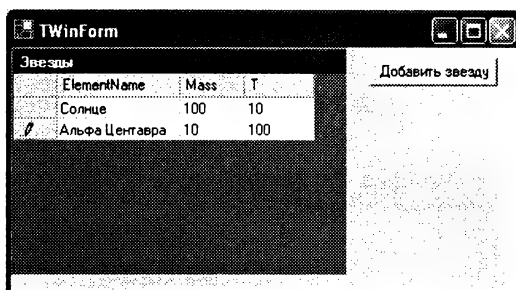


Рис. 9.7. Работающая программа ECO

Теперь программа полностью готова. Запустим ее и убедимся, что при нажатии на кнопку **Добавить звезду** в таблице автоматически появятся новые строки. В них можно вводить подходящие значения (рис. 9.7).

9.5.7. Построитель выражений OCL

Язык объектных ограничений OCL в данной книге не рассматривается. Тем не менее отметим наличие в системе Delphi удобного визуального построителя выражений OCL. В рассмотренном примере мы вручную вводили строку `Star.AllInstances` (отбор всех экземпляров класса `Star`) в свойство `Expression` дескриптора `ehStar`. Однако можно воспользоваться визуальным построителем выражений OCL.

Выражение `Star.AllInstances` можно сформировать визуально. Построитель выражений в текстовом поле `Expression` дескриптора `ehStar` вызывается мини-кнопкой с тремя точками, расположенной на правом краю этого поля. В левой верхней части построителя располагается формируемое выражение OCL.

В правой части окна построителя представлено дерево синтаксических элементов. Его содержимое динамически меняется в зависимости от текущей конструкции OCL. В исходном состоянии представляется дерево классов текущей модели (раздел `Classes`).



Дерево в правой части не сможет появиться, если предварительно не была выполнена компиляция проекта. Как уже говорилось, проект необходимо компилировать и сохранять после любых модификаций модели и связанных с ней компонентов, если в последующем необходим учет или использование каких-либо модельных функций.

Дважды щелкнем на элементе класса `Star`, экземпляры которого мы хотим сделать доступными через идентификатор `ehStar`, — в окне выражения OCL появится текст:

Star.

В правой части окна появится новая категория выбора: `Ocl operations`. Она содержит набор операций OCL, применимых к текущему выражению `Star` (рис. 9.8).

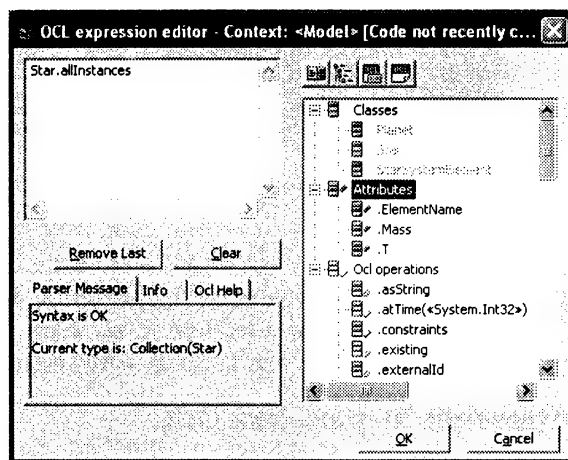


Рис. 9.8. Визуальный редактор выражений OCL

В списке доступных операций выберем пункт `.AllInstances` двойным щелчком. Выражение OCL примет вид:

`Star.AllInstances`

Нажмем кнопку `OK` — выражение запишется в свойство `Expression` объекта `ehStar`.

9.5.8. Расширяем возможности приложения ECO

Дополним форму приложения второй кнопкой. Введем в качестве ее заголовка строку `Добавить Планету`. Разместим также на форме вторую таблицу `DataGrid`. Пусть ее название (свойство `CaptionText`) — `Планеты`.

Добавим в проект новый компонент `ExpressionHandle` и назовем его `ehPlanet`. Он будет ответственен за доступ к экземплярам класса `Planet` объектного пространства. Введем в свойстве `RootHandle` значение `rhRoot` в качестве его корневого объекта. В свойстве `Expression` введем строку `Planet.AllInstances` (можно воспользоваться построителем выражений OCL).

Новая таблица представляется объектом `DataGrid2`. Свяжем ее через свойство `DataSource` с дескриптором `ehPlanet`. Для кнопки зададим связь с таблицей `DataGrid2` через ее свойство `BindingContext`, а в качестве корневого идентификатора в свойстве `RootHandle` выберем объект `Planet`. В результате в приложении будут функционировать две таблицы. Первая представляет звезды (экземпляры класса `Star`), а вторая — планеты (экземпляры класса `Planet`) — рис. 9.9.

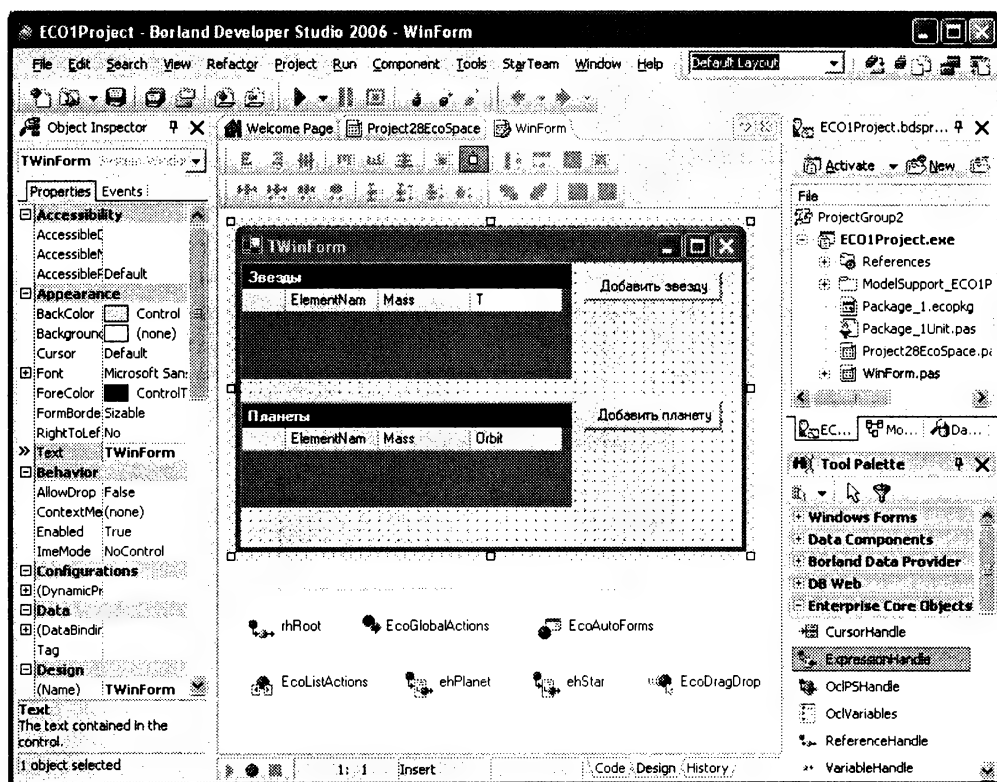


Рис. 9.9. Настраиваем таблицы и кнопки

В Инспекторе объектов имеются группы ECO | GUI и ECO | ListAction. В них сосредоточены свойства элементов управления, ответственные за связь с объектным пространством ECO. Мы уже познакомились с действием по добавлению нового объекта в это пространство. Кнопка выполнит такое действие, если в ее свойстве `EcoListAction` выбрано значение `Add`. В данном свойстве имеются также команды для перемещения текущего объекта по списку вверх (`MoveUp`) и вниз (`MoveDown`), для удаления текущего объекта ECO (`Delete`) и другие средства.

Добавим на форму новую кнопку и назовем ее Удалить планету. Свяжем ее с таблицей `DataGrid2`, визуализирующей экземпляры класса `Planet`, с помощью свойства кнопки `BindingContext`. В свойстве кнопки `RootHandle` укажем ссылку на дескриптор `ehPlanet`, связывающий форму с объектным пространством. А в свойстве `EcoListAction` кнопки выберем значение `Delete` (Удалить). Если теперь запустить программу, то будет возможно удалять строку, выделенную в таблице Планеты. При этом автоматически удалится соответствующий экземпляр класса `Planet`, расположенный в объектном пространстве.



При попытке удалить объект ECO из пользовательского интерфейса формы откроется стандартное диалоговое окно с запросом на подтверждение операции.

9.5.9. Автоформы

Свойству `EcoAutoForm` любой таблицы можно задать значение `True`. В результате содержимое таблицы станет доступно для редактирования в дополнительном режиме так называемых автоформ. При желании каждую из строк таблицы можно будет редактировать в отдельном окне. Достаточно дважды щелкнуть мышью на заголовке строки, как появится специальное диалоговое окно с полями ввода, соответствующими столбцам таблицы (рис. 9.10).

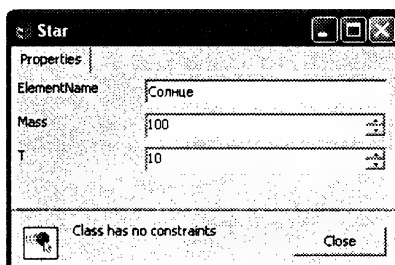


Рис. 9.10. Редактирование свойств объекта в автоформе

Для одной таблицы можно одновременно открыть сразу несколько окон автоформ. Данная возможность, как будет показано далее, особо полезна в случаях, когда между объектами таблицы установлены дополнительные связи.

9.6. Организация связей между объектами ECO

9.6.1. Добавляем ассоциативную связь

В реальных проектах взаимосвязи между объектами модели гораздо сложнее, нежели в нашем примере. Помимо связей вертикального наследования очень часто возникают горизонтальные взаимосвязи. Если «звезды» могут существовать в рамках нашей маленькой модели сами по себе, то каждая «планета» должна быть обязательно связана с определенной «звездой», вокруг которой она вращается.

Перейдем к модели, выделим на диаграмме UML элемент `Planet` и с помощью инструмента `Association` свяжем его с элементом `Star`. Так формируется ассоциативная связь между классами `Star` и `Planet`. Далее ее надо выделить и настроить с помощью Инспектора объектов.

В свойстве Name ассоциативной связи введем строку `ascToStar`. Под таким именем данная связь будет доступна в программе. Теперь надо указать, что эта связь направленная: ведь у одной звезды может быть произвольное число планет, а у одной планеты — только одна звезда. Для этого уточним значения свойств `End1` и `End2` связи `ascToStar`. Свойство `End1` соответствует стороне связи, относящейся к классу `Planet`, а свойство `End2` — стороне класса `Star`.

Дадим собственное имя стороне связи `End1`. В ее вложенном свойстве `Name` введем строку `Planets`. Кроме того, изменим вид связи стороны `End1`. В свойстве `Multiplicity` (Кратность) зададим форму отношения «многие к одному» — «0..*». Такая кратность означает, что одному экземпляру класса `Star` может соответствовать произвольное количество экземпляров класса `Planet`.

Откорректируем кратность также для стороны `End2`. Назначим для ее свойства `Multiplicity` значение 1. Это означает, что каждому экземпляру класса `Planet` может соответствовать один и только один экземпляр класса `Star`. Названием стороны `End2` (значением свойства `Name`) пусть будет строка `Stars` (рис. 9.11).

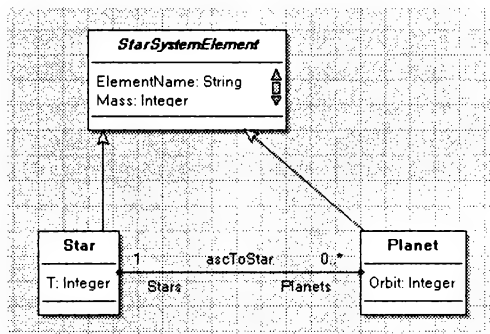


Рис. 9.11. Задаем ассоциативную связь между классами

9.6.2. Отслеживаем объект ЕСО, выбранный в таблице

На данный момент программа находится в несколько «разобранном» состоянии. Если ее запустить, то в таблице «Планеты» появится новое поле «Star». В нем должна храниться ссылка на связанный с текущей планетой экземпляр класса `Star`. Но поскольку такая связь явно не установлена, в поле отображается строка `<null>`. Она обозначает отсутствие связи. В свою очередь, в таблице «Звезды» новые поля не добавляются: ведь каждая звезда может быть связана со множеством планет одновременно, и простым добавлением полей здесь не обойтись.

Требуется, чтобы при нажатии на кнопку **Добавить планету** очередной объект `Planet` в таблице «Планеты» сразу создавался с уже настроенной ассоциативной связью. В качестве его «родительской» звезды должен быть авто-

матически выбран текущий объект из таблицы «Звезды». Им будет объект, соответствующий выделенной строке этой таблицы.

Для отслеживания выделенного объекта ECO предназначен компонент `CurrencyManagerHandle` группы `Enterprise Core Objects` палитры инструментов. Добавим его к проекту и укажем в свойстве `Name` название `cmhStar` (отслеживание объектов `Star`). В качестве корневого идентификатора в свойстве `RootHandle` зададим ссылку на дескриптор `ehStar`. Так выполняется привязка данного компонента к конкретному типу отслеживаемых объектов проекта. В качестве элемента, визуализирующего отслеживаемые объекты, в свойстве `BindingContext` объекта `cmhStar` зададим таблицу `DataGrid1` («Звезды»). Она позволяет выбрать конкретный элемент своего списка. Этой возможностью и воспользуется объект `cmhStar`.

Внесем дополнительные изменения в цепочку взаимосвязей между дескрипторами проекта. Новый объект `cmhStar` надо сделать корневым для дескриптора `ehPlanet`. Тогда через этот дескриптор будут доступны не все экземпляры класса `Planet`, а лишь те, что связаны с текущим элементом таблицы «Звезды». Для этого в свойстве `RootHandle` объекта `ehPlanet` выберем новое значение `cmhStar` (вместо старого `rhRoot`). В качестве выражения OCL в свойстве `Expression` объекта `ehPlanet` введем строку `self.Planets`. Здесь часть `self` означает ссылку на самого себя (на конкретный экземпляр класса `Planet`), а часть `.Planets` является названием одной из сторон ассоциативной связи `ascToStar`. Она обозначает свойство `Planets`, связывающее текущую звезду со множеством планет.



В редакторе выражений OCL доступ к нужной стороне ассоциативной связи (`Planets` или `Stars`) задается в разделе `Roles` дерева иерархии.

Запустив программу, обнаружим, что для каждой звезды (строки таблицы «Звезды») можно создать свой, уникальный набор планет (строк таблицы «Планеты»). При навигации по списку звезд в нижнем списке автоматически будут представляться планеты, соответствующие текущей звезде, выбранной в первой таблице. Планеты теперь представляются не общим списком, а по группам, согласно принадлежности конкретным звездам.

9.6.3. Применяем автоформы к связанным таблицам

Зададим обоим таблицам режим поддержки автоматических форм. Для этого в свойство `EcoAutoForm` каждой таблицы внесем значение `True`. Теперь записи таблиц можно редактировать в отдельном немодальном окне, которое открывается двойным щелчком на левой части записи в таблице. В полях редактирования автоформы представляются не только собственные свойства выбранного объекта, но и поля сторонних объектов, связанных с ним. В нашем случае при открытии в автоформе объекта таблицы «Звезды» на вкладке `Planets` (планеты, связанные с текущей звездой) будет доступна

отдельная таблица. Она позволяет вводить, редактировать и удалять «планеты» (рис. 9.12).



При использовании автоформ результаты манипуляций с пространством ECO синхронно отображаются во всех окнах и таблицах одновременно.

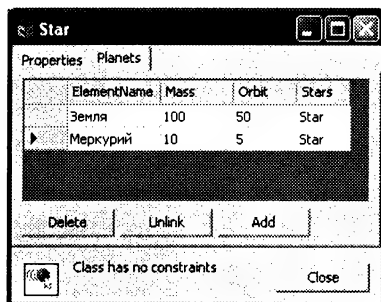


Рис. 9.12. Редактирование подчиненных объектов в автоформе

9.6.4. Настраиваем визуализируемые колонки

Запустим программу. В поле Stars таблицы «Планеты» представлено одно и то же значение — Star. Это не название конкретной звезды, а лишь наименование пристыкованного по ассоциативной связи класса, которое менять, очевидно, нельзя.

В таблице «Планеты» желательно в дополнение к неизменному названию класса отображать заданное пользователем название звезды, с которой текущая планета связана. Для этого следует настроить дескриптор ehPlanet.

Перейдем в Дизайнер, выделим в проекте элемент ehPlanet и обратимся к его свойству Columns (Доступные столбцы). В открывшемся диалоговом окне добавим дополнительные столбцы, которые будут доступны в таблицах и других элементах пользовательского интерфейса.

Новые столбцы добавляют кнопкой Add текущего диалогового окна. В кнопку встроен раскрывающийся список с двумя пунктами. Первый пункт — EventDerivedColumn — сформирует программный обработчик, в котором будет храниться код расчета нового виртуального поля. Второй пункт — OclColumn — позволяет сформировать значение в столбце с помощью выражения OCL. Выберем последний вариант. В результате в правой части текущего диалогового окна появится описание нового столбца в виде перечня его свойств. Нужно выражение формируется обращением к свойству Expression в этом перечне и вызовом построителя OCL.

Название нового столбца задается в свойстве Name. Введем в него строку StarName. В свойстве IsReadOnly (только чтение) введем значение True. Таким образом мы устанавливаем запрет на изменение названия звезды, связанной с текущей планетой, в таблице «Планеты».

В качестве выражения OCL в свойстве Expression введем строку `self.Stars.ElementName`. В ней происходит обращение к текущему экземпляру класса Planet (`self`). Этот элемент ссылается на единственный (в соответствии с кратностью связи) экземпляр класса Star, что задано частью выражения `.Stars` (название одной из сторон ассоциативной связи). Последним в выражении OCL записано обращение к свойству `ElementName` этого связанного объекта. Оно хранит название звезды, связанной с данной планетой (рис. 9.13).

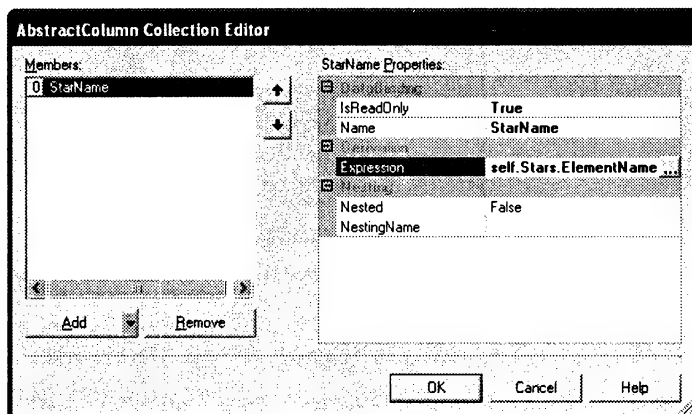


Рис. 9.13. Редактор колонок

Нажмем на кнопку OK, и в таблице «Звезды» в Дизайнере сразу же появится новый столбец `StarName`. Запустите программу и убедитесь, что в этом поле корректно отображается название родительской звезды (рис. 9.14).

9.7. Доступ к модели ECO на уровне исходных текстов

С помощью визуальных средств моделирования Delphi можно быстро создавать проекты с весьма сложной логикой. Платформенно-независимые сценарии, описывающие эту логику, формируются на универсальном языке OCL и встраиваются в диаграммы модели UML. Поэтому развитие и совершенствование проекта может происходить на высоком уровне абстракции (на уровне модели). Однако в практических проектах обычно требуется реализовывать более тонкие аспекты взаимодействия программы с человеком, нежели те, что предоставляют стандартные элементы управления, в частности таблицы `DataGrid`. Поэтому не всегда удастся избежать ручного программирования, да это и не требуется. Оптимальная схема разработки предполагает сочетание визуальных и моделирующих возможностей оболочки при построении общей архитектуры. На одних этапах работы удобно пользоваться средства-

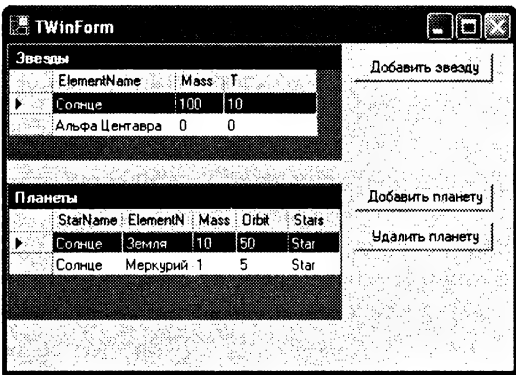


Рис. 9.14. Работающая программа ECO

ми быстрого создания типовых форм и привлекать стандартные средства взаимодействия с пользователем. На других этапах лучше прибегнуть к ручному кодированию характерных особенностей и оригинальных деталей проекта.

9.7.1. Программное создание объектов ECO

Реализуем вышеописанные возможности примера на уровне операторов языка Delphi. Заново создадим пустой проект ECO. Назовем его ECO3Project. Подготовим три описанных выше класса: StarSystemElement, Star и Planet с соответствующими атрибутами. Сформируем между ними начальные взаимосвязи наследования. На форме разместим кнопку Добавить звезду, а также таблицу DataGrid1. В ее свойстве CaptionText введем строку Звезды.

Для связи пользовательского интерфейса с объектным пространством добавим к проекту дескриптор (компонент ExpressionHandle). Назовем его ehStar, а в качестве родительского дескриптора (свойство RootHandle) зададим объект rhRoot. В качестве выражения OCL введем в свойство Expression строку Star.AllInstances. Назначим также связь таблицы DataGrid1 с дескриптором ehStar — через ее свойство DataSource.

В предыдущем примере создание и удаление объектов ECO обеспечивалось визуальной настройкой свойств кнопки Добавить звезду в Инспекторе объектов. Теперь мы вручную запрограммируем эти действия.

Создадим обработчик нажатия кнопки. В нем надо записать команду создания нового экземпляра класса Star. Новый объект ECO создается вызовом стандартного конструктора Create, доступного для всех классов ECO. Параметром его должна стать ссылка на объектное пространство ECO текущего приложения. Эта ссылка называется EcoSpace и представляет собой свойство текущей формы. Оно автоматически ей добавлено в процессе начального создания проекта ECO.

```

procedure TwinForm1.Button1_Click(sender: System.Object; e:
System.EventArgs);
begin
    Star.Create(EcoSpace);
end;

```

Данный код функционально эквивалентен визуальной настройке кнопки. Тогда мы задали в ее свойстве `RootHandle` дескриптор, связывающий кнопку с конкретным набором экземпляров объектного пространства. Выполняемое ею действие (создание объекта ECO) вводилось в свойстве `EcoListAction`. В программном коде достаточно создать экземпляр класса `Star`. Доступ к объектам, размещенным в пространстве ECO, осуществляется с помощью выражений OCL.

Новый экземпляр класса `Star` автоматически разместится в объектном пространстве, связь с которым организована через идентификатор `EcoSpace`. Доступ к экземплярам конкретного класса `Star` происходит через дескриптор `ehStar`. Этот дескриптор поставляет объекты таблицы «Звезды».

Запустим программу. После каждого нажатия на кнопку **Добавить звезду** таблица автоматически пополняется новым объектом.

Дополним проект второй таблицей «Планеты», второй кнопкой **Добавить планету** и новым дескриптором `ehPlanet`. Свяжем дескриптор с классом `Planet` через свойство `Expression` — введем выражение OCL `Planet.AllInstances`.

Обработчик нажатия на вторую кнопку запишется так:

```

procedure TwinForm1.Button2_Click(sender: System.Object; e:
System.EventArgs);
begin
    Planet.Create(EcoSpace);
end;

```

Теперь пользователю программы доступны две кнопки. Кнопка **Добавить звезду** добавляет объект в таблицу «Звезды», а кнопка **Добавить планету** добавляет объект в таблицу «Планеты».

9.7.2. Программное удаление объектов ECO

Удаление объекта ECO требует больших усилий, нежели его создание. Необходимо каким-то способом определить, какой конкретно элемент подлежит удалению.

Для отслеживания элемента, выбранного в таблице «Звезды», воспользуемся уже знакомым компонентом `CurrencyManagerHandle`, но в данном случае обратимся к нему программно. Поместим на форму новую кнопку и назовем ее **Удалить звезду**. Добавим в проект компонент `CurrencyManagerHandle` и назовем его `cmhStar`. Его свойство `RootHandle` настроим на дескриптор `ehStar`. Таким образом мы получим доступ к экземплярам класса `Star`. В свойстве `BindingContext` выберем таблицу `DataGrid1`. Значение свойства `RootHandle` объекта `ehPlanet` по умолчанию равно `rhRoot` (ссылка на корневой дескриптор). Заменим его на

значение `cmhStar`. В свойстве `Expression` введем строку `self.Planets`. Она содержит выражение OCL, описывающее список объектов ECO, доступных через текущий дескриптор `ehPlanet`. В результате этот дескриптор поставит в программу не все без исключения экземпляры класса `Planet`, а лишь связанные с текущим объектом, выбранным в таблице «Звезды».

Перед ликвидацией объекта необходимо проверить его тип. Объект, связанный с текущей строкой таблицы `DataGrid1`, должен быть представителем класса `Star`. В проверке типа поможет свойство `Element` компонента `cmhStar`. Оно имеет тип `IElement` и определяет программный интерфейс доступа к выбранному элементу. С помощью метода `AsObject` интерфейса `IElement` проверим тип текущего объекта, доступного через дескриптор `cmhStar`:

```
procedure TwinForm.Button3_Click(sender: System.Object; e:
System.EventArgs);
begin
    if cmhStar.Element.AsObject is Star then
        begin

            end;
        end;
```

Протестированный таким образом интерфейс описывает существующий экземпляр класса `Star`. Надо создать его локальную копию, интерфейс `IObject`¹ которой позволит напрямую манипулировать содержимым объектного пространства и удалить нужный экземпляр.

```
procedure TwinForm.Button3_Click(sender: System.Object; e:
System.EventArgs);

    // определяем локальный элемент:
    var SelStar: Star;
begin
    if cmhStar.Element.AsObject is Star then
        begin
            // готовим локальный объект
            // для доступа к пространству EcoSpace
            SelStar := Star( cmhStar.Element.AsObject );
            end;
        end;
```

Локальный объект `SelStar` создан на базе объекта, выделенного в таблице «Звезды» и доступного через дескриптор `cmhStar`. Задействуем интерфейс `IObject` объекта `SelStar` — он поставляется методом `AsObject`. Вызовем метод `Delete` для уничтожения выделенного в таблице объекта:

¹ Интерфейс `IObject` присущ всем объектам ECO. Он предназначен для доступа к внутренней структуре объекта в пространстве `EcoSpace`.

```

...
if cmhStar.Element.AsObject is Star then
begin
  SelStar := Star( cmhStar.Element.AsObject );
  SelStar.AsIOObject.Delete;
end;
...

```

Корпорация Borland рекомендует применять для удаления объектов ECO метод `Delete`, а не методы `Destroy` или `Dispose`. Объект ECO может находиться в одном из множества состояний, среди которых не только стандартные состояния «существует в памяти» и «удален из памяти», но и ряд более специфических. В частности, имеется состояние «полностью удален»: в него объект переводится вызовом метода `Delete`. При этом физически объект остается в объектном пространстве, так как он может быть связан с собственной копией в базе данных, которая тоже требует корректного удаления. Поэтому реальное удаление объектов из пространства `EcoSpace` вместе со всеми их копиями выполняет в подходящее время исполнительная система ECO. Программе достаточно перевести объект ECO в состояние «полностью удален» с вызовом его метода `Delete`.

9.7.3. Программное связывание объектов ECO

Теперь мы рассмотрим, как программно связывать друг с другом объекты, между классами которых установлена ассоциативная связь. Между классами `Star` и `Planet` на диаграмме модели установим ассоциативную связь с кратностью «один ко многим». Сторону связи, пристыкованную к классу `Star`, назовем `ToStar`, а противоположную сторону — `ToPlanet`. Названия сторон вводятся в свойстве `Name` вложенных свойств `End1` и `End2` ассоциативной связи (рис. 9.15).

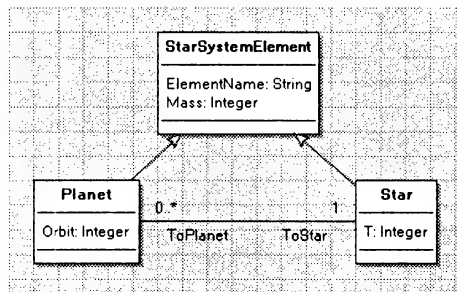


Рис. 9.15. Настройка ассоциативной связи

Разместим на форме кнопку `Добавить планету`. Никаких стандартных действий ECO в Инспекторе объектов для нее задавать не требуется. Подготовим

программный обработчик нажатия на эту кнопку. В нем создадим экземпляр класса `Planet` и свяжем его с текущим элементом таблицы «Звезды».

В обработчике выполняются следующие действия.

1. Проверяется соответствие типа выделенного в таблице элемента типу `Star`.
2. Создается новый экземпляр класса `Planet`. Он автоматически добавится в таблицу «Планеты».
3. Этот экземпляр связывается с элементом, выбранным в таблице «Звезды».

```
procedure TwinForm.Button2_Click(sender: System.Object; e:
System.EventArgs);
    var NewPlanet: Planet; // Новая создаваемая планета
        SelStar: Star; // Выбранная в таблице звезда
begin
    if cmhStar.Element.AsObject is Star then
        begin
            SelStar := Star( cmhStar.Element.AsObject );

            // создаем новую планету:
            NewPlanet := Planet.Create(EcoSpace);

            // формируем связь планеты со звездой:
            NewPlanet.ToStar := SelStar;
        end;
    end;
```

На последнем операторе остановимся поподробнее.

```
NewPlanet.ToStar := SelStar;
```

Свойство `ToStar` экземпляра класса `Planet` — это название стороны ассоциативной связи, стыкующей класс `Planet` с классом `Star` в соотношении «многие к одному». Оно имеется у каждого объекта класса `Planet` и было автоматически добавлено в этот класс после того, как мы сформировали данную связь. На диаграмме модели мы присвоили названия `ToStar` и `ToPlanet` двум концам ассоциативной связи с помощью Инспектора объектов. Программно же эта связь реализована в виде «оконечных» свойств — полей связанных классов. Сама по себе ассоциативная связь как отдельный объект в программе не существует.

Ассоциативное связывание объекта `SelStar` с объектом `NewPlanet` в программном коде реализуется с помощью обычного оператора присваивания. Связь настраивается автоматически с учетом кратности «один ко многим», заданной на диаграмме. Среда времени выполнения ЕСО в ходе такого динамического связывания выполнит настройки для обеих сторон ассоциативной связи. Объект `NewPlanet` свяжется с объектом `SelStar` через свое свойство `ToStar`, а свойство `ToPlanet` объекта `SelStar`, в свою очередь, получит в качестве значения ссылку на объект `NewPlanet`.

Вместо последнего оператора

```
NewPlanet.ToStar := SelStar;
```

можно записать противоположное действие

```
SelStar.Planets.Add(NewPlanet);
```

Итоговый результат окажется прежним в силу двунаправленности связей между объектами ЕСО. Только в данном случае выполняется не обычное присваивание, а вызывается метод `Add`. Ведь свойство `ToPlanet` хранит не скалярный элемент, а список узлов, связанных с объектом `SelStar`. Этот список отражает множественную часть ассоциативного отношения «многие к одному», поэтому обращаться к нему надо как к коллекции элементов.

9.8. Технология связи модели ЕСО с базой данных

Технология ЕСО позволяет автоматически связывать объектное пространство ЕСО с его копией на физическом носителе (например, с базой данных или файлом). Эта особенность технологии ЕСО крайне важна, поскольку она позволяет сохранять данные, введенные в таблицы пользовательского интерфейса, между сеансами работы с приложением.

Если мы не примем необходимых мер сохранения, то все данные, введенные в программу по ходу рассматриваемого примера, пропадут, как только программа будет закрыта. Лучше всего хранить содержимое объектного пространства в СУБД. Технология ЕСО обеспечивает такую возможность, что позволяет проектировать на уровне модели масштабные корпоративные приложения.

9.8.1. Принципы использования СУБД в технологии ЕСО

Связь объектного пространства ЕСО с внешним хранилищем выполняется в ходе следующих операций.

1. Формируется связь приложения с СУБД.
2. Задействуются компоненты объектно-реляционной раскладки ЕСО. Они стыкуют модель ЕСО с компонентами приложения, обеспечивающими связь с базой данных.
3. В базе данных генерируется набор таблиц, предназначенных для хранения копии объектного пространства.
4. Пользовательский интерфейс расширяется элементами, синхронизирующими содержимое объектного пространства с его копией на внешнем носителе.

Рассмотрим эти операции более подробно.

9.8.2. Связываем приложение ЕСО с СУБД

Вернемся к первому примеру, в котором мы разработали приложение ЕСО только с помощью визуальных средств. Перейдем на закладку рабочего окна

Delphi под названием `ECO1ProjectEcoSpace` (предполагается, что название текущего проекта — `ECO1Project`). В открывшемся окне размещаются компоненты, связывающие объектное пространство с внешними хранилищами данных. Исходно оно пусто, поскольку объектное пространство работает стандартным способом и его содержимое нигде не сохраняется.

Выберем на палитре инструментов компонент `BdpConnection` из категории `Borland Data Provider` и поместим его в текущее окно. Этот компонент предназначен для установки связи с СУБД. Настроим его на любую доступную базу данных. В качестве учебного примера выберем какую-либо из демонстрационных баз СУБД `Interbase`, например базу `employee.gdb`. В свойстве `ConnectionString` компонента `BdpConnection` зададим название соответствующего соединения (например, `IBConn1`).

9.8.3. Используем компоненты объектно-реляционной раскладки

Следующий шаг — настройка автоматической связи модели ECO с избранной СУБД. Синхронизация содержимого пространства ECO с данными на внешних носителях (файлах или базах данных) выполняется компонентами, наследующими базовые характеристики класса `PersistenceMapper`. Этот класс ответственен за отображение (раскладку) структуры объектного пространства в различные схемы представления данных.

Выберем компонент `PersistenceMapperBdp` из категории `Enterprise Core Objects` палитры инструментов и поместим его в окно `ECO1ProjectEcoSpace`. Этот компонент предназначен для отображения структуры объектного пространства в реляционные схемы баз данных, доступ к которым происходит по технологии `BDP`. Экземпляр созданного объекта автоматически получит название `PersistenceMapperBdp1` и расположится в дизайнерском пространстве окна рядом с объектом `BdpConnection1` (экземпляром компонента `BdpConnection`).

Свяжем объект `PersistenceMapperBdp1` с объектом `BdpConnection1`, выбрав значение `BdpConnection1` в свойстве `Connection`.

Компонент `PersistenceMapperBdp` полностью ответственен за автоматическое сохранение и загрузку копии объектного пространства модели в выбранной базе данных. Фактически, мы реализовали основные аспекты взаимодействия приложения с базой данных, просто добавив компонент `PersistenceMapperBdp` к проекту (рис. 9.16).

9.8.4. Генерируем схему базы данных

Копия объектного пространства хранится в наборе специальных взаимосвязанных таблиц, расположенных в выбранной базе данных. Внутренняя структура этих таблиц и связи между ними формируются при помощи компонента `PersistenceMapperBdp`. Эта структура соответствует на физическом уровне формальной модели приложения, заданной диаграммами `UML`. В компонент `PersistenceMapperBdp` встроены средства автоматической трансляции этой моде-



Рис. 9.16. Настройка компонента связи с СУБД

ли в сценарий на языке обработки реляционных таблиц SQL. В ходе работы сценарий и создаст нужные таблицы.

Механизмы трансляции модели приложения UML в сценарий SQL существуют в разных системах моделирования. Имеются они и в среде Delphi. Однако реализации языка SQL в различных СУБД существенно различаются. Порой даже основные операторы SQL в разных СУБД работают по-разному. Поэтому разработчик предварительно должен настроить процесс генерации структуры базы данных на конкретную технологию СУБД.

Настройки этого процесса хранятся в свойстве `SqlDatabaseConfig` компонента `PersistenceMapperBdp`. Оно состоит из множества вложенных подсвойств, определяющих способы записи команд SQL. Эти подсвойства можно настраивать вручную, а можно автоматически. В последнем случае надо выбрать подходящий пункт контекстного меню компонента `PersistenceMapperBdp`. При использовании СУБД Interbase нужный пункт называется `Interbase [dialect3] setup`, а, например, при использовании СУБД Microsoft SQL Server — `SQL Server setup` (рис. 9.17).

В нашем примере мы выбираем пункт `Interbase [dialect3] setup`. В результате все подсвойства свойства `SqlDatabaseConfig` компонента `PersistenceMapperBdp` будут корректно настроены на версию языка SQL для СУБД Interbase.

Выполним генерацию таблиц, описывающих структуру текущего объектного пространства. Для этого надо нажать кнопку `Generate Schema` (Генерировать схему) в нижней части окна пространства ECO. В результате будет сгенери-



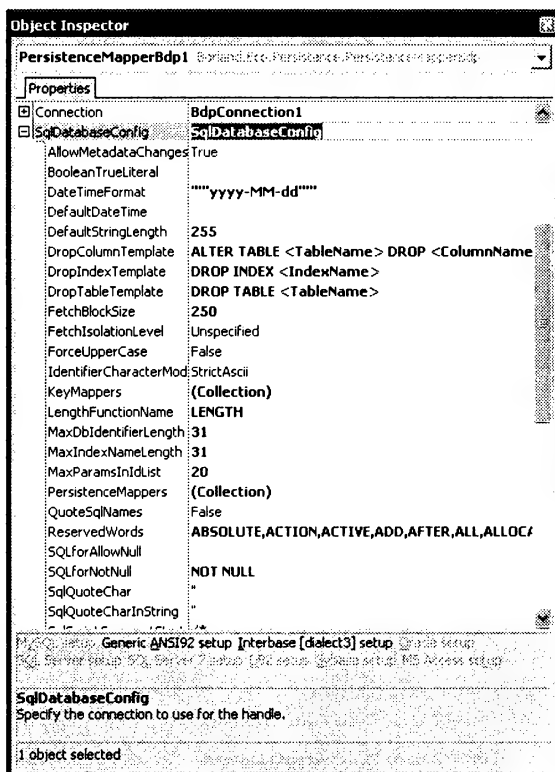


Рис. 9.17. Настройка диалекта SQL

рован сценарий SQL для конкретной технологии СУБД. Исполнение сценария запустится автоматически, и в текущей базе данных будут созданы все таблицы и взаимосвязи между ними, сохраняющие копию объектного пространства.

В ходе генерации таблиц среда Delphi проверит содержимое выбранной базы данных. Если в ней обнаружатся таблицы, не имеющие отношения к текущей модели, среда предложит их удалить. По умолчанию режим удаления лишних таблиц отключен. Если выбрана тестовая база данных employee.gdb с наборами демонстрационных примеров из поставки СУБД Interbase, то удалять из нее другие таблицы, конечно, не следует. В реальном проекте для хранения модели проекта лучше заранее создать отдельную базу данных.

Список таблиц, предлагаемых для удаления, показан в диалоговом окне процесса генерации на вкладке Optionally Delete. Если в базе имеются таблицы, которые ранее уже создавались для других проектов ECO, то они будут явно рекомендованы для удаления. На вкладке Required drop/recreate (Требуется пересоздать) выводится список таблиц, которые будут удалены и затем сно-

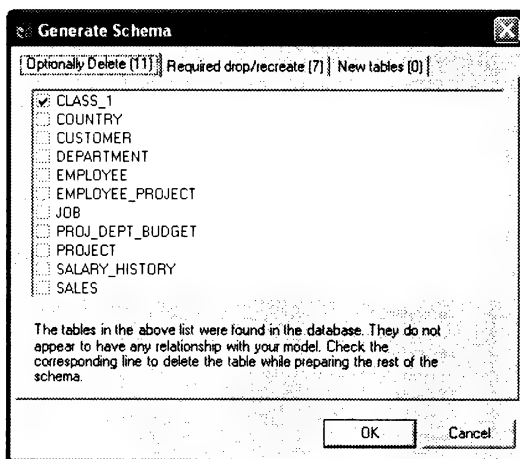


Рис. 9.18. Список автоматически создающихся таблиц

ва заново воссозданы. Список новых таблиц приведен в разделе New tables (рис. 9.18).

Нажмем кнопку ОК, и в окне сообщений (Messages) среды Delphi будет представлен протокол генерации и сообщение об успешном завершении процесса Schema Generation! Database schema created (рис. 9.19).

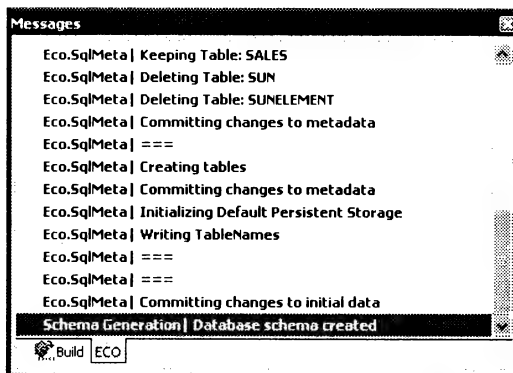


Рис. 9.19. Протокол генерации схемы базы данных

Мы настроили компонент PersistenceMapperBdp на передачу данных между объектным пространством ECO и СУБД. Теперь надо связать приложение с этим компонентом. Щелкните мышью на пустом месте окна ECO1ProjectEcoSpace в Дизайнере. В свойстве PersistenceMapper (разворачивающийся список) Инспектора объектов выберите объект PersistenceMapperBdp1. Теперь связь объектного пространства приложения с СУБД установлена.

9.8.5. Расширяем интерфейс пользователя

Согласование содержимого базы данных с содержимым объектного пространства выполняется вызовом метода `UpdateDatabase` объекта `EcoSpace` — глобальной ссылки приложения на текущее объектное пространство. Будем вызывать этот метод новой кнопкой с заголовком Обновить. Обработчик нажатия на эту кнопку запишется так:

```
procedure TWinForm.Button5_Click(sender: System.Object; e:
System.EventArgs);
begin
    EcoSpace.UpdateDatabase;
end;
```

В единственной команде `EcoSpace.UpdateDatabase` обработчика вызывает-ся метод `UpdateDatabase` текущего пространства ECO. Он выполняет полную синхронизацию содержимого пространства ECO в оперативной памяти с его копией в базе данных.

Запустим программу, введем несколько описаний звезд, для каждой из них зададим планеты со своими характеристиками. Запись этой информации в базу данных произойдет после нажатия на кнопку Обновить. Когда программа запустится в следующий раз, в таблицы «Звезды» и «Планеты» автоматически загрузится содержимое объектного пространства, которое было сохране-но в базе данных в момент последнего выполнения команды `UpdateDatabase`.

9.8.6. Синхронизация модели и базы данных: визуальная настройка

Синхронизацию содержимого пространства ECO и базы данных можно выполнять, не прибегая к ручному кодированию. Делается это так. Выберем в Дизайнере кнопку Обновить и удалим ее программный обработчик из кода программы. Обратимся к свойству `EcoAction` кнопки — оно хранит перечень стандартных действий технологии ECO. Выберем в разворачивающемся списке стандартное действие `UpdateDatabase`. Когда пользователь нажмет на эту кнопку, приложение сохранит в базе данных содержимое объектного пространства. Кнопка во время работы приложения автоматически меняет свое состояние. Если нет необходимости обновлять пространство ECO (никаких изменений и рассогласований не зафиксировано), она будет заблокирована.

9.8.7. Использование файлов для хранения пространства ECO

Технология ECO позволяет хранить объектное пространство не только в базе данных, но и в обычных файлах. Структура модели и содержимое объектов ECO записывается в файл в виде данных в формате XML. Компонент, связывающий пространство ECO с файлами, называется `PersistenceMapperXml`.

Разместим компонент `PersistenceMapperXml` в окне настройки объектного пространства `ECO1ProjectEcoSpace` текущего проекта. В свойстве `PersistenceMapper` этого пространства записано значение `PersistenceMapperBdp1` — ссылка на объект связи приложения с СУБД. Изменим значение этого свойства на ссылку `PersistenceMapperXml1`.



Объект `PersistenceMapperBdp1` удалять из проекта не обязательно. Он просто временно отключен от проекта.

В свойстве `FileName` объекта `PersistenceMapperXml1` введем название файла (например, `model.xml`), в котором будет храниться содержимое объектного пространства.



Работа с файлами значительно менее надежна и эффективна, нежели с СУБД, которые поддерживают транзакции, контролируют целостность данных и берут на себя еще целый ряд функций, неподвластных драйверу файлов XML. Если создается многопользовательское приложение ECO, крайне желательно хранить копию объектного пространства в базе данных, а не в файле.

9.8.8. Множественные соединения приложения ECO с СУБД

В одном проекте ECO допускается несколько соединений объектного пространства с различными СУБД. Программа переключается между ними динамически, непосредственно во время работы. Подходящие компоненты (наследники класса `PersistenceMapper`), ответственные за новые соединения, размещаются в окне `ECO1ProjectEcoSpace`. Каждый из них может быть настроен на стыковку с совершенно разными СУБД: MS SQL Server; Borland Interbase; DB2 — или, например, на взаимодействие с файлом XML.

Конкретный объект, связывающий пространство ECO с некоторой СУБД, задается в свойстве `PersistenceMapper` окна `ECO1ProjectEcoSpace`. Если в качестве значения этого свойства выбран пункт (`none`), копия объектного пространства сохраняться нигде не будет.

Добавим к проекту компоненты `PersistenceMapperBdp` и `BdpConnection`. Настроим их на связь приложения с одной из доступных СУБД. Таким образом, в текущем проекте окажутся два компонента объектно-реляционной раскладки — `PersistenceMapperBdp` и `PersistenceMapperXML`.

Динамическое переключение между этими компонентами надо запрограммировать вручную. Для этого необходимо, прежде всего, изменить видимость данных компонентов в приложении. Введем в свойство `Modifiers` каждого из этих компонентов значение `public`.

Пусть в зависимости от некоторого условия требуется сохранять копию объектного пространства либо в СУБД, либо в файле XML. Такую возмож-

ность полезно предусмотреть, в частности, в задачах периодического архивирования данных. Соответствующий код в основной программе запишется так:

```
...
if условие
then EcoSpace.PersistenceMapper :=
    EcoSpace.PersistenceMapperBdp1
else EcoSpace.PersistenceMapper :=
    EcoSpace.PersistenceMapperXml1 ;
EcoSpace.UpdateDatabase;
...
```

Оператор присваивания настраивает пространство EcoSpace на связь с конкретным объектом — промежуточным звеном между приложением и хранилищем данных. Команда UpdateDatabase непосредственно обновляет содержимое пространства ECO в доступной базе данных или файле XML.

Пусть, например, на форме находится флажок Checkbox1 с заголовком Сохранять в базе данных. Если он включен, то задействуется объект PersistenceMapperBdp1. В противном случае пространство ECO сохранится в файле XML:

```
procedure TForm1.Button1_Click(sender: System.Object; e:
System.EventArgs);
begin
    if Checkbox1.Checked
    then EcoSpace.PersistenceMapper :=
        EcoSpace.PersistenceMapperBdp1
    else EcoSpace.PersistenceMapper :=
        EcoSpace.PersistenceMapperXml1 ;
    EcoSpace.UpdateDatabase;
end;
```

9.8.9. Технология создания модели ECO на основе существующей базы данных

Технология ECO позволяет сохранять в базе данных копию объектного пространства ECO, построенного на основе модели UML. Она также способна выполнять обратную задачу — формировать модель приложения на языке UML. Эта модель строится автоматически, путем анализа существующей базы данных. Процесс формирования модели выполняется специальным Мастером, который становится доступен после настройки объектного пространства проекта на связь с конкретной СУБД.

Создадим пустое приложение ECO. На закладке пространства ECO (окно Project1EcoSpace) разместим компонент связи с СУБД BdpConnection и компонент объектно-реляционной раскладки PersistenceMapperBdp. Настроим их на выбранную СУБД и свяжем друг с другом, как было описано выше.

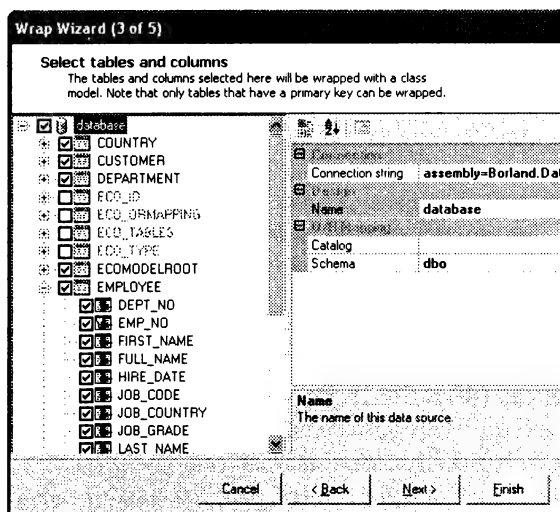


Рис. 9.20. Выбор таблиц и полей для построения модели

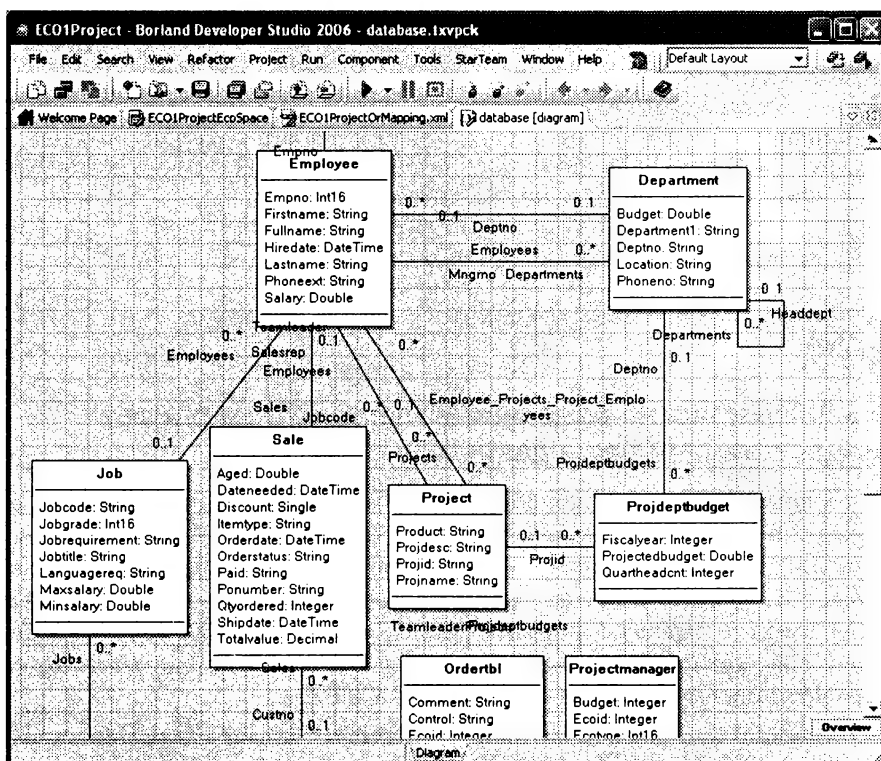


Рис. 9.21. Модель, автоматически построенная на основе базы данных

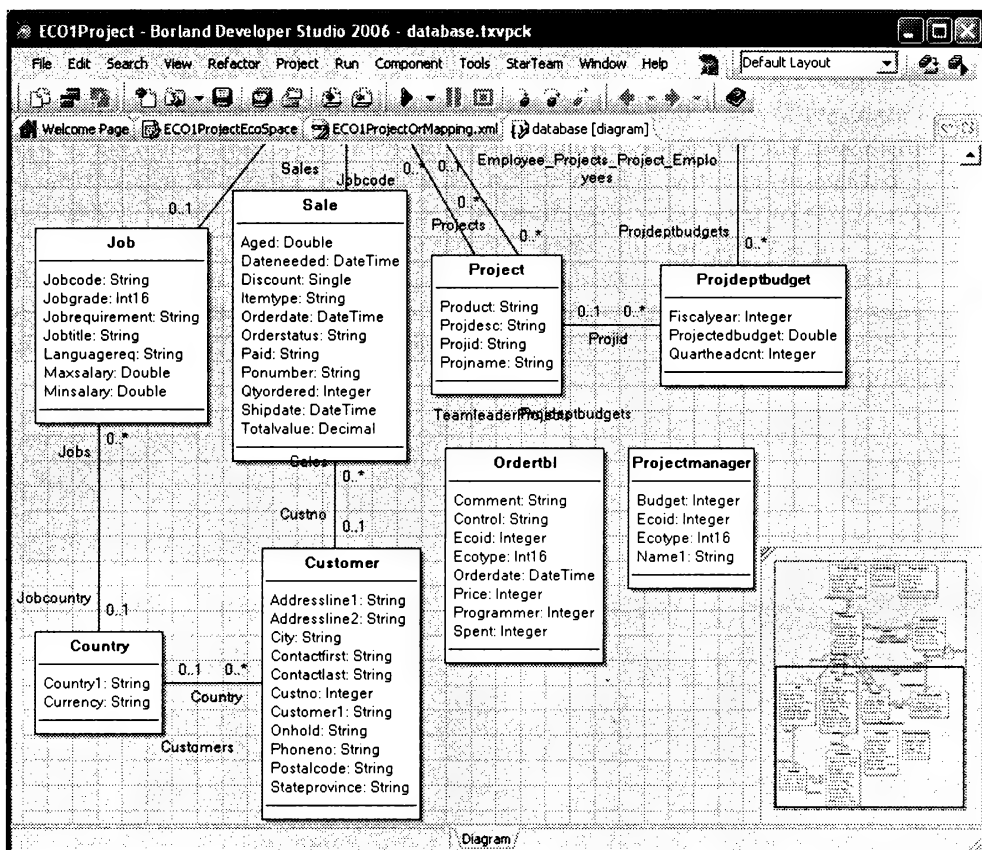


Рис. 9.22. Средство просмотра всей модели

Нажмем на кнопку **Wrap Existing Database with Eco** (Построить модель ECO на основе существующей базы данных) в нижней части окна **Project1EcoSpace**. На экране появится Мастер формирования модели ECO. Выберем, например, демонстрационную базу данных **employee.gdb** из поставки СУБД **Interbase**, которой мы уже неоднократно пользовались в других примерах. Тогда вариант преобразования базы будет выглядеть в Мастере примерно так, как показано на рис. 9.20.

Флажками в окне Мастера помечены таблицы, описания которых будут использованы при построении модели. Нажмем на кнопку **Next** — Мастер сгенерирует файл XML, в который запишется описание анализируемой базы данных. Каждой таблице в этом описании отведен свой класс, а между ними будут установлены правильные взаимосвязи. Новое модельное пространство доступно в проекте в виде пакета, имя которого вводилось на первом шаге работы Мастера.

«Сырая» модель, извлеченная из базы данных, обычно выглядит не очень приглядно. В частности, образы таблиц из базы данных и линии связей между ними располагаются на диаграмме хаотически. Команда контекстного меню диаграммы Layout ➤ Do Full Layout (Раскладка ➤ Выполнить полную раскладку) автоматически упорядочит расположение элементов и взаимосвязей (рис. 9.21).

В нижнем правом углу окна модели имеется поле Overview. Если на нем щелкнуть, то откроется окно, представляющее всю модель целиком. Желаемую область просмотра можно задать с помощью мыши (рис. 9.23).

Мастер преобразования базы данных по вышеописанной технологии формирует готовую модель некоторой понятийной области. На основе этой автоматически созданной модели не представляет труда создать полноценное приложение ECO.

Алфавитный указатель

A

Abstract Factory, шаблон проектирования 146
action, действие 232
activity, деятельность 232
Activity Diagram, диаграмма деятельности 224
Adapter, шаблон проектирования 161
ADO.NET, технология связи с СУБД 91
Artifact, элемент 237
ASP.NET, Web-технология 30, 113
association, ассоциативное отношение 202

B

BDP.NET, технология связи с СУБД 91
BDP Pool Manager, менеджер пула связи 95
Borland Together, среда моделирования 30
Borland UDDI Browser, браузер 37
Bridge, шаблон проектирования 163
Builder, шаблон проектирования 150

C

CaliberRM, среда управления требованиями 26
Cassini, Web-сервер 127

Chain of Responsibility, шаблон проектирования 175
Change Parameters 59
Class Diagram, диаграмма классов 197
CLR, среда времени выполнения 113
Code Folding, средство рефакторинга 48
Code Insight, режим подсказок 44
Code Sync, режим синхронного редактирования 50
Code Templates, шаблоны кода 46
Collaboration Diagram, диаграмма кооперации 218
Command, шаблон проектирования 177
Communicate Diagram, диаграмма коммуникации 240, 242
Component Diagram, диаграмма компонентов 226
Composite, шаблон проектирования 165
Composite Structure Diagram, диаграмма внутренней структуры 239

D

DataSnap, технология многоуровневых систем 108
Data Explorer, проводник данных 37
Data Set, набор данных 93

- dbGo, набор компонентов для связи с СУБД 100
- DB Web, технология создания Web-приложений 124
- Declare New Field, средство рефакторинга 57
- Declare Variable, средство рефакторинга 57
- Decorator, шаблон проектирования 167
- Delegation Connector, связь 237
- dependency, отношение зависимости 206
- Deployment Diagram, диаграмма развертывания 228
- Deployment Manager, менеджер развертывания 114
- Designer Guidelines, режим проектирования формы 34
- Device, элемент UML 237
- DLLImport, технология импорта функций 82
- E**
- ECO III, технология моделирования 30
- EcoSpace, объектное пространство 245
- Execution Environment, элемент UML 237
- Extract Method, средство рефакторинга 58
- Extract Resource String, средство рефакторинга 62
- Extract Superclass, средство рефакторинга 57
- F**
- Facade, шаблон проектирования 170
- Factory Method, шаблон проектирования 153
- FastMM4, менеджер памяти, 25
- Find Class, средство рефакторинга 56
- Find Reference, средство рефакторинга 56
- Find Unit, средство рефакторинга 55
- Flyweight, шаблон проектирования 171
- G**
- generalization, отношение обобщения 205
- Global Assembly Cache, кэш .NET 37
- H**
- HTML-дизайнер 35
- I**
- Inline Variable, средство рефакторинга 61
- Interpreter, шаблон проектирования 179
- Introduce Field/Variable, средство рефакторинга 60
- Iterator, шаблон проектирования 180
- L**
- Live Designer, режим проектирования 34
- M**
- Mediator, шаблон проектирования 182
- Memento, шаблон проектирования 183
- Model View, средство просмотра 37, 42
- Move, средство рефакторинга 61
- MSIL, язык низкого уровня 85
- O**
- Object Repository, репозиторий объектов 31

Observer, шаблон проектирования
185

P

«pan and scroll», технология
горизонтального
скроллинга 39

Platform Invoke, сервис
дистанционного вызова методов
82

Prototype, шаблон проектирования
156

Provided Interface, предоставляемый
интерфейс 235

Proxu, шаблон проектирования 174

Pull Members Down, средство
рефакторинга 61

Pull Members Up, средство
рефакторинга 61

R

realization, отношение реализации
202

Rename, средство рефакторинга 59

Required Interface, запрашиваемый
интерфейс 235

S

Safe Delete, средство рефакторинга
62

Sequence Diagram, диаграмма
последовательностей 213, 240

Simple Object Access Protocol
(SOAP), протокол Web-служб
129

Singleton, шаблон проектирования
159

Smart Block, режим рефакторинга
48

snippet, шаблон 47

SQL-запрос 102
формирование 102

State, шаблон проектирования 186

State Machine Diagram, диаграмма
машин состояний 242

Statechart Diagram, диаграмма
состояний 221

Strategy, шаблон проектирования
188

Structure View, просмотрщик
структуры 40

Sync Edit, режим синхронного
редактирования 40

T

Template Method, шаблон
проектирования 190

TFlowPanel, плавающая панель 39

TGridPanel, панель-таблица 39

Tidy, средство Web-
форматирования 35

ToDo, средство ведения
напоминающих записей 52

U

Unicode, кодировка 85

Unified Modeling Language (UML),
унифицированный язык
моделирования 195

V

VCL-форма
создание 34

Virtual Library Interface, сервис
виртуальных библиотек 82

Visitor, шаблон проектирования
191

W

Web-калькулятор 117

Web-сервис
организация доступа 138

Web-служба 129
интерфейс 135
клиент 135
обращение 141

Web-служба (*продолжение*)

разработка 129

создание 130

структура 130

Web-форма 115

проектирование 116

Web Services Description Language (WSDL), язык описания Web-службы 129

Х

XML-файл 125

загрузка данных 127

синхронизация 127

сохранение данных 126

А

автовычисление 45

автозавершение 44

автозаполнение 45

автокоррекция 45

автомат 221

вложенный 223

состояние 222

автоописание 45

автосправка 45

автоформа 258

редактирование 258

Агрегатор 181

адаптер 95

Актер 208

артефакт 237

архитектура

многозвенная 104

многоуровневая 104

атрибут класса 80

дополнительный 81

пользовательский 82

создание 81

Б

база данных

dbGo 100

адрес 123

внутренняя структура 90

генерация 269

схема 269

быстрое комментирование 53

быстрое перемещение строки в ресурсы 62

В

визуализация кода 42

визуальная настройка 273

визуальное создание метода 58

визуальное создание суперкласса 57

вызов

асинхронный 241

синхронный 241

Г

генерация таблиц 270

гиперсвязь 230

гиперссылка 230

создание 230

глобальное переименование идентификатора 59

глобальный кэш сборок 37

Д

действие 232

дескриптор ЕСО 246

деятельность 232

диаграмма

варианты использования 208, 238

взаимодействия 231

внешняя 241

внутренней структуры 239

деятельности 224, 232

диаграмма (*продолжение*)
 класса 197
 коммуникаций 240, 242
 компонентов 226, 236
 кооперации 218
 последовательности 213, 240
 прецедентов 208
 развертывания 228
 составная 239
 состояний 221, 242
 экспорт 229
Дизайнер 33
 режим работ 33
Диспетчер шаблонов 193

З

закладка 50
 удаление 50

И

идентификатор
 корневой 252
 локальный 56
 необъявленный 57
 поиск 56
иерархическая структура кода 40
Инспектор объектов 36
интерфейс 207
 Web-службы 135
 абстрактный 146
 Агрегатор 181
 виртуальной библиотеки 82
 запрошенный 235
 компоненты 39
 пользовательский 253
 поставляемый 235
 продуктовый 153
 элементы 254
исключение 85
 обработка 85

источник данных 93
 абстрактный 102
 гетерогенный 104
 физический 93

К

категория 31
клавиатурные макросы 54
 воспроизведение 54
класс
 абстрактный 68, 199, 248
 ассоциаций 204
 атрибуты 80, 199
 внутренний тип 70
 добавление на диаграмму 198
 дочерний 61
 закрытый 68
 иерархия 251
 константа 70
 кратность 203
 поиск 56
 поле 69, 199
 родительский 61
 свойства 199
классификатор 232
клиентское приложение 106
ключевой элемент 46
кнопка фильтрации 32
код
 ассемблерный 86
 визуализация 42
 неуправляемый 82
 просмотр 86
 смешанный 85
 управляемый 82
комментарий 228
коммуникационный путь 238
компонент 226
конфигуратор 95

корневой идентификатор 252
кратность 200
дискретная 236

М

массив
 .NET 80
 двумерный 72
 динамический 72
 многомерный 72
 описание 80
 размещаемый 80
 статический 73
Мастер преобразования базы
 данных 278
Мастер установки 26
Мастер формирования модели
 ЕСО 277
Мастер шаблонов 145
машина состояний 30
Менеджер BDP-пула 95
Менеджер проектов 36
Менеджер развертывания 114
Менеджер шаблонов 46
метод 201
 абстрактный 202
 безопасное удаление 62
 интерпретации 179
 параметры 58
 перебора элементов 180
 статический 69
миниредатор коллекции 102
многоуровневое приложение 103
модуль удаленного доступа к базе
 данных 109

Н

набор данных 93
 типизированный 93
 формирование 93

напоминания 52
 добавление 52

О

обновление модели 42
 автоматическое 42
объект 207, 214
 внутренняя структура 88
 добавление 207
 название 207
объект ЕСО 246
 программное связывание 266
 программное создание 263
 программное удаление 263
объект-вектор 75
объектно-реляционная раскладка
 269
операция
 перегрузка 74
описатель видимости 200
опорная сетка 33
 настройка 34
 отображение 33
 параметры 34
 шаг 33
отладка 84
 дистанционная 87
 пошаговая 89
отношение 202
 агрегации 203
 ассоциации 202, 210
 включения 212
 зависимости 206
 композиции 204
 обобщения 205, 211
 расширения 211
 реализации 202

П

палитра инструментов 31

- панель-сетка 39
- паттерн проектирования 145
- перемещение описания между классами 61
- перемещение поля внутрь класса 60
- перемещение членов между классами 61
- плавающая панель 39
- подавтомат 223
- подсказка
 - поиск 53
 - сортировка 53
 - фильтрация 53
- подсостояние 242
- поиск класса 56
- поиск ссылок 56
- поле класса 69
- помощники классов 68
- порт 235
- посредник 136
- поставщик данных 90, 101
 - выбор 101
 - дистанционный 111
 - текущий 105
- построитель выражений OCL 255
- поток данных 234
- поток управления 233
- преобразование типов 77
 - неявное 78
 - перегрузка 77
 - явное 77
- прерывание 84
 - системное 84
 - список 84
- приложение ASP.NET 114
 - отладка 127
 - соединение с базами данных 118
 - создание 115
 - структура 114

- приложение BDP.NET 93
 - многоуровневое 104
- приложение ЕСО 247
 - множественные соединения с СУБД 274
 - связь с СУБД 268
 - создание 247, 275
- Проводник данных 37, 90
- прокси-класс 136
- Просмотрщик моделей 37
- Просмотрщик структуры программы 40
- пространство имен 66
- протокол
 - генерации 272
 - отладки 84
- процедура
 - встраиваемая 71
 - вызов 98
 - настройка 99
 - тестовый запуск 98
 - храняемая 98
- пул соединений 95

Р

- расширенный набор символов 64
- расширенные комментарии 53
- редактор
 - Delphi 40
 - кода 35
 - HTML 117
 - коллекции поставщиков данных 105
 - параметров 59
 - соединений 94
 - формирования SQL-запросов 36
- режим
 - автоввода 32
 - автоматического сворачивания компонентов 31

режим (*продолжение*)
автоматического притягивания
элементов 33
автоматической подсказки 44
быстрого завершения описания
48
вертикального расположения
заголовков 31
визуальный 90
направляющих линий 34
отображения подсказок 44
поддержки автоформ 260
предварительного просмотра 63
проектирования формы 253
размещения групп компонентов
31
синхронного редактирования 50
форматирования 35
Репозиторий объектов 31
доступ 31
рефакторинг 55
отмена 63

С

связь ассоциативная 204, 258
добавление 258
настройка 258
связь с базами данных 124
связывание данных 120
сервер приложений 105
доступ 111
регистрация 110
создание 105, 108
серверное приложение 118
символическая таблица 87
загрузка 87
синтаксическая корректность 40
синхронизация 273
система форматирования
документов 35

совместная деятельность 239
соединение
активное 102
создание 90
проверка 94
сообщение 215
асинхронное 216
обычное 216
синхронное 216
спецификаторы видимости 67
список отложенных дел 52
список шаблонов кода 47
среда исполнения 237
средства компактного свертывания
кода 48
средство просмотра структуры 41
средство модельного
представления 43
сторожевое условие 222
суперсостояние 242
схема наследования 197

Т

таблица
загрузка 87
перенос содержимого 100
принимающая 100
связанная 120
символическая 87
создание 91
структура 91, 120
схема 100
удаление 91
тип данных 79
базовый 79
точка декларации 40
точка прерывания 84

У

удаление лишних переменных 61

- узел 228
 - сворачивания кода 49
- уровень
 - middleware 104
 - клиентского приложения 104
 - серверных модулей 104
 - СУБД 104

Ф

- фокус активности 215
- форма 33
- фрейм взаимодействия 240

Ц

- цветовая схема 31

Ш

- шаблон проектирования 145
 - Абстрактная фабрика 146
 - Адаптер 161
 - группа 145
 - Декоратор 167
 - Интерпретатор 179
 - Итератор 180
 - Команда 177
 - Компоновщик 165
 - Мост 163
 - Наблюдатель 185
 - Обслуживание 193
 - Одиночка 159
 - поведенческий 145
 - порождающий 145
 - Посетитель 191

- шаблон (*продолжение*)

- Посредник 182
- Приспособленец 171
- Прокси 174
- Прототип 156
- Состояние 186
- Стратегия 188
- Строитель 150
- структурный 145
- Фабричный метод 153
- Фасад 170
- Хранитель 183
- Цепочка обязанностей 175
- Шаблонный метод 190

Э

- элемент
 - автоматическое удаление 62
 - ветвления 224
 - интерфейса 254
 - контейнера 73
 - оформления панели 31
 - быстрое создание 60
 - тематическая группа 31
 - цвет 45

Я

- язык
 - моделирования UML 195
 - объектных ограничений OCL 255
 - описания Web-служб (WSDL) 129

Бобровский Сергей Игоревич

Технологии Delphi 2006. Новые возможности

Заведующий редакцией
Ведущий редактор
Научный редактор
Литературный редактор
Художник обложки
Корректор
Верстка

*А. Кривцов
Л. Панич
Г. Евсеев
С. Симонович
Л. Адуевская
О. Голотвина
И. Симонович*

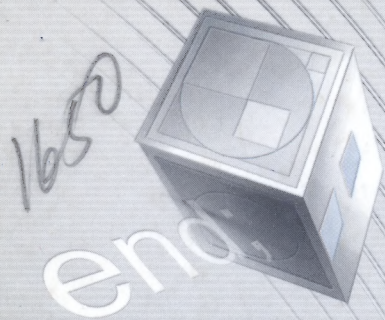
Подписано в печать 03.05.06. Формат 70×100/16. Усл. п. л. 23,22.

Тираж 3000 экз. Заказ № 1331.

ООО «Питер Пресс», 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.

Налоговая льгота — общероссийский классификатор продукции
ОК 005-93, том 2; 953005 — литература учебная.

Отпечатано с диапозитивов в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.



Сергей Бобровский

ТЕХНОЛОГИИ DELPHI 2006

Новые возможности

Большинство нововведений версии Delphi 2006, рассмотренных в книге, связано с поддержкой технологий платформы Microsoft .NET. Разработчику доступны как средства создания стандартных приложений Windows Forms .NET, так и средства построения приложений .NET на базе библиотеки визуальных компонентов Borland VCL.NET. Система Delphi предлагает также обширный набор технологий для создания программ в классической архитектуре Win32. Особое внимание в книге уделено новым возможностям оболочки Delphi и средствам рефакторинга.

В издании представлены средства Delphi 2006 по визуальному формированию приложений, взаимодействующих с СУБД с помощью провайдеров BDP.NET, описано создание серверных приложений ASP.NET, веб-служб и клиентских программ для них.

Последние версии системы Delphi активно дополняются новыми технологиями проектирования программ. Книга рассказывает о шаблонах проектирования для языка Delphi, о средствах Delphi по созданию диаграмм UML 1.5 и 2.0 и о новой версии технологии моделирования ECO III. Читатель познакомится с созданием приложений ECO III на уровне модели, узнает, как готовить диаграммы UML, которые автоматически транслируются в исходные тексты на языке Delphi, изучит механизмы ECO по организации объектно-реляционной раскладки.

- Ускорение разработки за счет новых средств рефакторинга и шаблонов проектирования
- Визуальные средства взаимодействия с СУБД с помощью провайдеров BDP.NET
- Программирование приложений ASP.NET и веб-служб на языке Delphi
- Создание корпоративных приложений ECO на базе моделей UML

НОВЫЕ ВОЗМОЖНОСТИ — НОВЫЕ ПЕРСПЕКТИВЫ

Тема: Программирование/Delphi

Уровень пользователя: опытный/эксперт

ISBN 5-469-01438-X



9 785469 014386

ПИТЕР®

Заказ книг:

197198, Санкт-Петербург, а/я 619
тел.: (812) 703-73-74, postbook@piter.com

61093, Харьков-93, а/я 9130
тел.: (057) 712-27-05, piter@kharkov.piter.com

www.piter.com — вся информация о книгах и веб-магазин